FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Ereignisbasierte Leistungsanalyse von Remote-Memory-Access-Operationen

Marc-André Hermanns

FZJ-ZAM-IB-2004-15

Dezember 2004

(letzte Änderung: 16. Februar 2005)

Inhaltsverzeichnis

1 Einleitung **3**

2 Ereignisbasierte Leistungsanalyse **5**
2.1 Motivation . . . 5
2.2 Leistungsanalysezyklus . . . 6
2.2.1 Instrumentierung . . . 6
2.2.2 Analyse der Ereignisdaten und Präsentation der Ergebnisse . . . 10
2.2.3 Optimierung der Anwendung . . . 11
2.3 Ereignisbasierte Leistungsanalyse mit KOJAK . . . 11
2.3.1 Statische Instrumentierung mit OPARI und EPILOG . . . 12
2.3.2 Analyse des Ereignisstroms mit EARL und EXPERT . . . 13
2.3.3 Präsentation der Analyseergebnisse durch CUBE . . . 17
2.3.4 Darstellung des Ereignisstroms durch VAMPIR . . . 18

3 MPI-2: Einseitige Kommunikation **19**
3.1 Motivation . . . 19
3.2 Fenster als Rahmen für den Datenaustausch . . . 20
3.3 Datenübertragung . . . 21
3.4 Synchronisation . . . 22
3.4.1 Synchronisation mit aktivem Ziel . . . 23
3.4.2 Synchronisation mit passivem Ziel . . . 27
3.5 Korrekter RMA-Zugriff . . . 29
3.6 Assertions - Einflussnahme auf die Synchronisation . . . 31

4 Ereignismodelle **33**
4.1 Motivation . . . 33
4.2 Einführung in die Ereignismodelle . . . 34
4.3 Ereignistypen der einseitigen Kommunikation . . . 39
4.4 Entwicklung der Ereignismodelle . . . 41
4.4.1 Das Basismodell . . . 42
4.4.2 Das erweiterte Modell . . . 45
4.4.3 Modellierungsaspekte der Sperren bei den vorgestellten Modellen . . . 49

5 Erweiterung von KOJAK **51**
5.1 Motivation . . . 51
5.2 Änderungen für die Messphase . . . 52
5.2.1 Wrapper . . . 52
5.2.2 Fensterverwaltung . . . 53
5.2.3 Gruppenverwaltung . . . 56
5.2.4 Datentransfer . . . 58
5.2.5 Sperren . . . 60

5.3 Zusammenführung der Einzelspuren 61
5.3.1 Basismodell: Keine Veränderung der Aufzeichnungszeitpunkte 61
5.3.2 Erweitertes Modell: Verschiebung der Datentransferendpunkte . . . 64
5.4 Änderungen für die Analysephase 66

6 Leistungsmerkmale **71**
6.1 Einleitung 71
6.2 Einfache Leistungsmerkmale 73
6.2.1 MPI Communication 74
6.2.2 MPI Synchronization 74
6.3 Musterbasierte Leistungsmerkmale 75

7 Visualisierung **79**
7.1 Motivation 79
7.2 Geschlossene Visualisierung der RMA-Operationen 79
7.3 Visualisierung des logischen Datentransfers 81
7.4 Visualisierung der Abhängigkeiten 83

8 Zusammenfassung und Ausblick **87**

A Glossar **89**

Literaturverzeichnis **95**

Abbildungsverzeichnis

2.1 Leistungsanalysezyklus 7
2.2 Ausnutzung einer schwachen Symbolbindung 10
2.3 KOJAK-Architektur und die Schnittstelle zu VAMPIR 12
2.4 Ereignistypen in EARL 2.0 14
2.5 Leistungsmerkmale und Leistungsprobleme in EXPERT 3.0 16
2.6 Dreigeteiltes Präsentationsfenster von CUBE 17

3.1 Synchronisation mit Fence 24
3.2 Schemata der allgemeinen Synchronisation mit aktivem Ziel 25
3.3 Schemata der Synchronisation mit passivem Ziel 28
3.4 Logisches Schema für korrekte RMA-Zugriffe bei MPI 30

4.1 Modellierung von Regionen und resultierender Aufrufbaum 37
4.2 Modellierung der blockierenden und nicht-blockierenden P2P-Kommunikation 39
4.3 Synchronisation mit Fence im Basismodell 42
4.4 Allgemeine Synchronisation mit aktivem Ziel im Basismodell 43
4.5 Synchronisation mit passivem Ziel im Basismodell 44
4.6 Synchronisation mit Fence im erweiterten Modell 45
4.7 Allgemeine Synchronisation mit aktivem Ziel im erweiterten Modell 46
4.8 Synchronisation mit passivem Ziel im erweiterten Modell 47
4.9 Aufzeichnung der allgemeinen Synchronisation mit aktivem Ziel 48
4.10 Lösungsansatz für die Modellierung der Sperren 49

5.1 Schema des ersten Durchlaufs der Zusammenführung 62
5.2 Schema des zweiten Durchlaufs der Zusammenführung 64
5.3 Schema der Warteschlangen-Datenstruktur des erweiterten Modells 65
5.4 Erweiterte Struktur der Ereignistypen in EARL 2.1 67

6.1 Hierarchie der Leistungsmerkmale in EXPERT 3.1 72
6.2 Das Leistungsproblem Wait at Fence 76
6.3 Das Leistungsproblem Late All Complete bzw. Early Wait 76
6.4 Das Leistungsproblem Late Complete 76

7.1 Geschlossene Visualisierung der RMA-Operationen 80
7.2 Visualisierung des logischen Datentransfers 82
7.3 Visualisierung der Abhängigkeiten 84

Tabellenverzeichnis

5.1 Aufbau des ELG_MPI_WIN-Definitionseintrags 54
5.2 Aufbau des ELG_MPI_WINEXIT-Ereigniseintrags 55
5.4 Aufbau des ELG_MPI_COMM-Definitionseintrags 56
5.3 Aufbau des ELG_MPI_WINCOLLEXIT-Ereigniseintrags 56
5.5 Aufbau des ELG_MPI_PUT_1TS-Ereigniseintrags 58
5.6 Aufbau des ELG_MPI_PUT_1TE- und ELG_MPI_PUT_1TE_REMOTE-Ereigniseintrags 59
5.7 Aufbau des ELG_MPI_GET_1TO-Ereigniseintrags 59
5.8 Aufbau des ELG_MPI_GET_1TS- und ELG_MPI_GET_1TS_REMOTE-Ereigniseintrags 59
5.9 Aufbau des ELG_MPI_GET_1TE-Ereigniseintrags 60
5.10 Aufbau des ELG_MPI_WIN_LOCK-Ereigniseintrags 60
5.11 Aufbau des ELG_MPI_WIN_UNLOCK-Ereigniseintrags 61

Zusammenfassung

Um die Korrektheit und die Leistung eines Programms zu analysieren, braucht man Informationen über den dynamischen Ablauf aller beteiligten Prozesse. Dieser Ablauf kann als Ereignisstrom modelliert werden, bei dem alle für die spätere Analyse wichtigen Ereignisse und deren Attribute erfasst werden. Diese Art der Leistungsanalyse nennt man ereignisbasiert.

Eine am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich entwickelte Werkzeugumgebung zur Leistungsanalyse von parallelen Programmen (KOJAK) instrumentiert und analysiert neben OpenMP-Konstrukten auch MPI-Aufrufe des MPI-Standard 1.2. MPI-2 führt eine standardisierte Schnittstelle zur einseitigen Kommunikation (one-sided communication) bzw. zum entfernten Speicherzugriff (remote memory access) ein.

Remote-Memory-Access (RMA) beschreibt die Funktionalität eines Prozesses, direkt auf einen Speicherbereich eines anderen Prozesses zuzugreifen, ohne dass der entfernte Prozess am Datenaustausch explizit beteiligt ist. Damit unterscheidet sich diese Kommunikation zum Beispiel von der Punkt-zu-Punkt-Kommunikation, bei der dies über Send- und Receive-Aufrufe in den jeweiligen Prozessen getätigt wird. Da die Parameter des Datenaustausches nur von einem der beiden Kommunikationspartner bestimmt werden, spricht man auch von *einseitiger Kommunikation*.

Die Diplomarbeit beschäftigt sich mit der Einbindung der Analyse der einseitigen Kommunikation in die bestehende ereignisbasierte Leistungsanalyse-Umgebung. Den Schwerpunkt bildet die Entwicklung eines Ereignismodells, welches die Abläufe der einseitigen Kommunikation möglichst realitätsnah im Ereignisstrom abbildet. Dieses beinhaltet vor allem die verschiedenen Synchronisationsschemata des MPI-Standards und deren Eigenschaften. Ferner werden die Instrumentierung der relevanten Funktionen des MPI-Standards und darüber hinaus auch einiger plattformabhängiger RMA-Schnittstellen, die aber wegen ihrer Einfachheit als Untermenge der für den MPI-Standard benötigten Modelle angesehen werden können, realisiert. Schliesslich werden Leistungsmerkmale, welche mit Hilfe der neuen Ereignisse nachgewiesen werden können, definiert und beschrieben.

Abstract

To analyse the correctness and the performance of a program, information about the dynamic behaviour of all participating processes is needed. This dynamic behaviour can be modeled as a stream of events that contains all events and their attributes required for a later analysis. This kind of performance analysis is called event-based.

KOJAK, a toolkit for performance analysis, developed at the Central Institute for Applied Mathematics of the Research Centre Juelich, instruments and analyses OpenMP statements and MPI calls of MPI 1.2. MPI-2 introduced a standardized interface to one-sided communication and remote memory access.

Remote memory access (rma) describes the functionality of a process, to directly access a part of the memory of a remote process, without explicit participation of the remote procces in the data transfer. This distinguishes the one-sided communication from other communication concepts, such as point-to-point communication, where explicit send and receive statements on the corresponding processes are required. As all parameters for the data transfer are determined by one process, it is also called *one-sided* or *single-sided* communication.

This thesis deals with the integration of performance analysis methods for one-sided communication into the existing analysis toolkit. Special emphasis is put on the development of an event model that realisticly represents the dynamic behaviour in the event stream. This must especially include all synchronisation methods described in MPI 2.0 and their characteristics. Additionally, other hardware specific rma interfaces, that form a subset to the model due to their simplicity, are covered. Finally, new performance properties based on the new events are described and defined.

Kapitel 1

Einleitung

Programmierer von Anwendungen mit nachrichtenbasierter Kommunikation sehen sich vor dem immer größer werdenden Problem, den Leistungsengpass ihrer Anwendung zu finden und zu verstehen. Dieses Problem ist oft in der Menge der gesammelten Leistungsdaten und dem für traditionelle Leistungsanalyse-Werkzeuge nötigen Expertenwissen begründet. Daher arbeiten Werkzeugentwickler verstärkt an Möglichkeiten, Anwendungen automatisch zu analysieren. Es wird nach Methoden gesucht, die gesammelten Leistungsdaten effizient zu analysieren. Der Anwender soll direkt zu den Stellen seines Programms dirigiert werden, die ineffizientes Verhalten aufweisen.

Gleichzeitig zu den Arbeiten an der Leistungsanalyse vervollständigen immer mehr Hersteller ihre MPI Implementationen bezüglich der MPI-2-Erweiterungen. Damit steht auch die einseitige Kommunikation, als eine dieser Erweiterungen, auf immer mehr Plattformen zur Verfügung. Es ist zu erwarten, dass Anwendungen nun verstärkt dieses neue Konzept nutzen. Somit wird auch der Bedarf an Werkzeugen zur effizienten Analyse dieser Anwendungen steigen.

Paradyn [6] und KOJAK [7, 20] sind zwei Beispiele für automatische Leistungsanalyse-Werkzeuge. Paradyn sammelt während der Laufzeit summarisch Daten über den Programmablauf. Durch die Zusammenfassung der Daten noch während der Aufzeichnung wird zwar weniger Speicherplatz verbraucht, aber es gehen auch Informationen über spezielle Ereignisinstanzen verloren. Eine spätere Analyse kann somit nur rein statistische Aussagen treffen, ohne zum Beispiel auf konkrete Instanzen eines Funktionsaufrufs eingehen zu können. KOJAK zeichnet zur Laufzeit eines Programms eine prozesslokale Abfolge von Ereignissen, die den Ablauf des Programms beschreiben, auf. Diese Einzelspuren werden *post mortem*, d.h. nach Beendigung des Programms, zu einer globalen Abfolge der aufgezeichneten Ereignisse zusammengefügt. Einzelne Ereignisinstanzen sind im Ereignisstrom vorhanden. Bei der Auswertung des Ereignisstroms sind somit Daten über einzelne Ereignisse verfügbar, die bei Paradyn aufgrund der Zusammenfassung verloren gegangen sind. Innerhalb der Analyse kann so einfacher nach Problemen im Zusammenspiel der einzelnen Prozesse gesucht werden. Das Sammeln dieser detaillierten Daten über den Programmablauf geht allerdings zu Lasten des Speicherplatzes und kann stärkeren Einfluss auf den Ablauf des Programms nehmen, als das von Paradyn benutzte Analysekonzept.

Automatische Leistungsanalyse von MPI-Remote-Memory-Access-Operationen ist bisher noch nicht weit verbreitet. Zur Supercomputing 2004 im November stellten Kathryn Mohror und Karen L. Karavanic eine Erweiterung des Paradyn Projektes vor [8]. Darin beschreiben sie erste Ansätze zur automatischen Leistungsanalyse und Leistungsengpässe, die bei der einseitigen Kommunikation mit MPI auftreten können.

Ziel dieser Arbeit ist es, das bestehende Werkzeug KOJAK um die Fähigkeit zu erweitern, Programmabläufe der einseitigen Kommunikation aufzuzeichnen und die Grundlage

zu schaffen, diese zu analysieren und Stellen für ineffizientes Laufzeitverhalten aufzuzeigen.

Im folgenden Kapitel 2 werden die Grundlagen der ereignisbasierten Leistungsanalyse besprochen. Es wird auf die Architektur von KOJAK eingegangen und die Zuordnung der einzelnen Bestandteile zu den verschiedenen Phasen der Leistungsanalyse eines Programms vorgenommen. Zusätzlich werden verschiedene Analysemethoden erläutert. Grundlegende Begriffe werden definiert, um in den weiteren Kapitel darauf aufbauen zu können. Ansätze des Profiling und des Tracings werden besprochen und ihre Vor- und Nachteile aufgeführt. Da KOJAK zu den spurbasierten Analyseprogrammen gehört, wird dort ein besonderer Schwerpunkt liegen, um in den weiterführenden Kapiteln die Erweiterungen bezüglich der einseitigen Kommunikation entwickeln zu können. Dabei wird auch die Zusammensetzung von KOJAK als Sammlung von verschiedenen Werkzeugen erklärt und auf die unterschiedlichen Stadien der Leistungsanalyse eingegangen. Die Erklärungen beschränken sich auf die für die Entwicklung der Analyse von RMA-Operationen relevanten Teile.

Kapitel 3 gibt eine Einführung in das Kommunikationsparadigma der einseitigen Kommunikation mit MPI. Dabei wird neben den zur Verfügung stehenden RMA-Operationen ein Fokus auf den möglichen Synchronisationsverfahren liegen. Dies bildet die Grundlage der in Kapitel 4 entwickelten Modelle in Kapitel 4. Ziel ist es schließlich, den Begriff der Abgeschlossenheit einer RMA-Operation innerhalb einer MPI-Umgebung zu definieren.

In Kapitel 4 werden verschiedene Modelle entwickelt, welche die Synchronisation der RMA-Operationen und eventuelle Abhängigkeiten im Ereignisstrom der Spur darstellen. Es wird ein bereits in dem kommerziellen Werkzeug VAMPIR/VAMPIRTRACE bestehendes Modell aufgegriffen und weiterentwickelt, um den Anforderungen der einseitigen Kommunikation gerecht zu werden. Weiterhin werden einzelne Leistungseigenschaften definiert, die bei der Verwendung der einseitigen Kommunikation auftreten können.

In Kapitel 5 wird die technische Seite der Erweiterung zur Analyse der einseitigen Kommunikation aufgeführt, wobei der Fokus auf den Erweiterungen zur Modell-Implementierung liegt. Die darin beschriebenen Veränderungen innerhalb der unteren Schichten der KOJAK-Architektur ermöglichen in Kapitel 6 die Definition neuer Leistungsmerkmale.

In Kapitel 7 werden Ansätze zur Visualisierung der Ereignisströme der einseitigen Kommunikation und ihr Einfluss auf das zugrundeliegende Ereignismodell erklärt. Da die Visualisierung von Ereignisströmen, insbesondere eine Darstellung der Ereignisse als Zeitlinien-Diagramm, in dem Bereich der manuellen Analyse fällt, werden die einzelnen Aspekte der Visualisierungsmodelle nicht in aller Ausführlichkeit behandelt. Es werden vielmehr die Möglichkeiten der Visualisierung mit dem benutzten Ereignismodell und die nötigen Änderungen zum Ereignismodell für weitere den Anwender unterstützende Darstellungen gegeben.

Schließlich gibt Kapitel 8 eine Zusammenfassung über die Ergebnisse der Arbeit und einen Ausblick auf die folgenden Schritte innerhalb des KOJAK-Projektes.

Da es für die im englischen MPI-Standard verwendeten Kernbegriffe der einseitigen Kommunikation keine standardisierte Übersetzung gibt, sind die in dieser Arbeit benutzten Begriffe im Anhang A ab Seite 89 mit einer kurzen Erklärung angegeben.

Kapitel 2

Ereignisbasierte Leistungsanalyse

2.1 Motivation

Bei der computergestützten Forschung werden die zu berechnenden Aufgaben immer komplexer. Um den ständig wachsenden Anforderungen an die Leistung moderner Computersysteme gerecht zu werden, besitzen Hoch- und Höchstleistungsrechner mehrere Prozessoren. Dadurch lassen sich bestimmte Aufgaben parallel und somit in geringerer Zeit berechnen. Man unterscheidet die Rechner-Architekturen SMP (shared memory processors[1]) und DMP (distributed memory processors).

Bei der SMP-Architektur können mehrere Prozessoren auf den gleichen gemeinsamen physikalischen Speicher zugreifen und darüber Daten eines anderen Prozesses manipulieren. Dafür wurde mit OpenMP eine standardisierte Schnittstelle geschaffen [1]. Diese Schnittstelle erlaubt die Aufteilung eines Prozesses in einzelne Threads und jedem dieser Threads Zugriff auf den gemeinsamen Prozessspeicher. Bei einer DMP-Architektur besitzt jeder Prozessor seinen eigenen lokalen Speicher. Prozessoren können nicht direkt entfernte Speicherstellen manipulieren, sondern tauschen Daten über Nachrichten aus. Diese Art des Datenaustauschs wird im Englischen *message passing* genannt und wurde als das Message Passing Interface (MPI) standardisiert [9, 10]. Beide Architekturen haben Vor- und Nachteile. Deshalb bilden moderne Computersysteme oft eine Mischung aus SMP und DMP. Dabei werden mehrere SMP Knoten in einem größeren DMP Verbund zusammengeschaltet. Hoßfeld et al. nennen diese hybriden Rechner gekoppelte SMP-Systeme [4].

Mit der Größe der Systeme und der Anzahl der kommunizierenden Prozesse wächst auch die Komplexität der Abläufe innerhalb eines parallelen Programms. Damit steigen die Anforderungen an den Anwender, diese Abläufe zu verstehen. Um die Leistung eines parallelen Programms bewerten zu können, müssen zunächst die Abläufe innerhalb des Programms aufgezeichnet werden. Eine manuelle Analyse solcher Leistungsdaten benötigt viel Erfahrung seitens des Anwenders. Durch die Komplexität der interagierenden Prozesse und die Anzahl der zu beachtenden Ereignisse werden die Mengen der gesammelten Leistungsdaten oft zu umfangreich, um selbst von erfahrenen Anwendern überblickt werden zu können. Etablierte Werkzeuge zur manuellen Analyse, wie das kommerzielle Produkt VAMPIR [3], unterstützen den Anwender bisher lediglich in der Visualisierung der Leistungsdaten, aber selten in der Interpretation. Um für die Leistungsanalyse mehr Produktivität zurückzugewinnen, versuchen Forschungsprojekte den Großteil der Analyse einem Programm zu überlassen und die Ergebnisse mit dem Fokus auf die eigentlichen Leistungsprobleme zu filtern.

Eine Methode, den Ablauf eines Programms zu modellieren, ist die ereignisbasierte Leis-

[1] oder symmetric multi processors

tungsanalyse. Hierbei werden das Auftreten und die Reihenfolge von Ereignissen betrachtet, die auf die Leistung des Programms Einfluss nehmen können. Ein Ereignis bezeichnet einen Punkt im Prozessablauf, der für die Leistungsanalyse wichtige Informationen beinhaltet. Da es sich um einen Zeitpunkt innerhalb des Programmablaufs handelt, besitzt er keine zeitliche Ausdehnung. Der Zustand eines Programms lässt sich durch die chronologische Abfolge aller bisher aufgetretenen Ereignisse modellieren. Dabei können Zustände sich aus anderen Zuständen ableiten. Der aktuelle Zustand eines Programms bestimmt sich aus der Gesamtheit der aktuellen Zustände aller Prozesse.

Das folgende Kapitel beschäftigt sich mit den einzelnen Schritten bei der Leistungsanalyse von Programmen, dem *Leistungsanalysezyklus*. Methoden zur Ereignisaufzeichnung werden vorgestellt. Bei der Erklärung der Analyse und Präsentation von Leistungsdaten wird auf die Werkzeugumgebung KOJAK und deren Arbeitsweise eingegangen. Das Kapitel dient als Grundlage zum Verständnis der ereignisbasierten Leistungsanalyse, insbesondere der in KOJAK enthaltenen Funktionalitäten.

2.2 Leistungsanalysezyklus

Die Analyse und Optimierung einer Anwendung durchläuft verschiedene Abschnitte, die in den nachfolgenden Kapiteln noch genauer beschrieben werden. Während der so genannten *Instrumentierungsphase* werden Instruktionen zur Ereignisaufzeichnung in den Programmcode eingefügt. Ein Lauf des instrumentierten Programms führt zur *Messung* der Daten. Diese Daten können zwischengespeichert oder direkt analysiert werden. Wenn die Analyse der Daten aufwendig ist, schließt sich die *Analysephase* erst nach Beendigung des Programms (*post mortem*) an. Bei geringer Beeinflussung des Programmablaufs kann die Analyse auch schon zur Laufzeit (online) durchgeführt werden. Nach der Analysephase folgen die Auswertung und die *Präsentation* der Analyseergebnisse. Daraus lassen sich Schritte zur *Optimierung* ableiten. Falls keine Leistungsprobleme festgestellt werden, wird die Instrumentierung des Codes wieder entfernt, um die volle Leistungsfähigkeit des Programms wiederzuerlangen. Im Falle notwendiger Optimierungen, gibt der Anwender das optimierte Programm wiederum in den Leistungsanalysezyklus ein, um die positiven Einflüsse der Änderungen am Programm zu verifizieren. An dieser Stelle werden dann auch Effekte sichtbar, bei denen sich mehrere Leistungsprobleme gegenseitig beeinflussen. D.h. Leistungsprobleme können auch dadurch auftreten, dass ein anderes Leistungsproblem an anderer Stelle existiert. Gleichzeitig beschreibt ein in der Leistungsanalyse gefundenes Leistungsproblem nur Symptome innerhalb des Laufzeitverhaltens eines Programms. Die Ursachen für die gefundenen Leistungsprobleme können in anderen Teilen des Programms verborgen sein. Eine Optimierung in mehreren Schritten ist deshalb sinnvoll. Der Leistungsanalysezyklus ist in Abbildung 2.1 schematisch dargestellt.

2.2.1 Instrumentierung

Die Ereignisse, die die Grunddaten zur ereignisbasierten Leistungsanalyse eines Programms bilden, müssen durch zusätzliche Instruktionen im Programmcode erzeugt werden. Durch zusätzliche Instruktionen im Programmcode ist es möglich, Informationen über die aktuellen Vorgänge im Programm zu speichern, um diese auf ineffizientes Verhalten zu analysieren. Diese Informationen werden in der ereignisbasierten Leistungsanalyse *Ereignisse* genannt. Diese Ereignisse werden explizit von dem zu messenden Programm selbst erzeugt. Innerhalb der ereignisbasierten Leistungsanalyse gibt es verschiedene Möglichkeiten diese Daten zu erheben und während der Laufzeit zu behandeln.

Um den Unterschied zu anderen Verfahren der Erfassung von Laufzeitinformationen in der

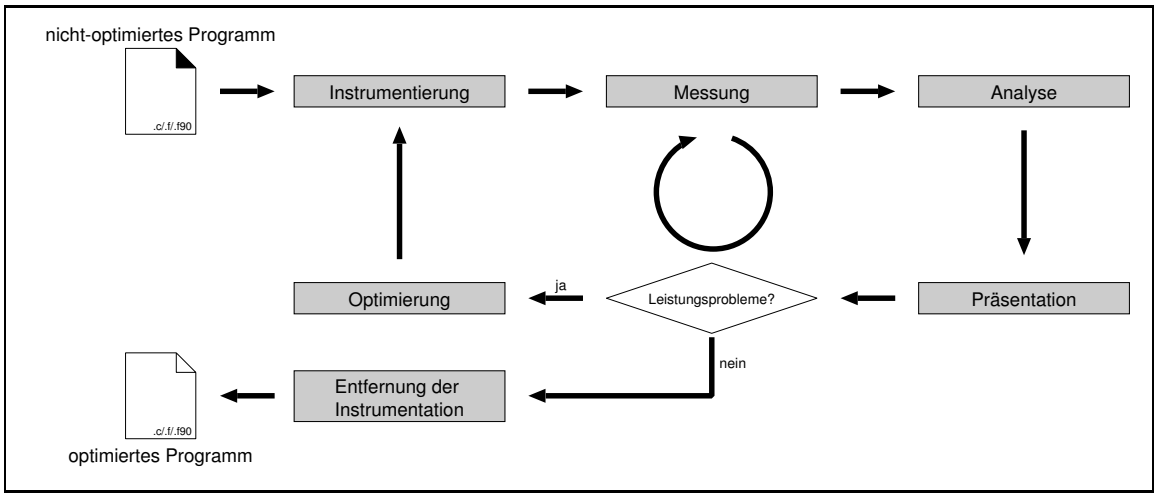


Abbildung 2.1: Leistungsanalysezyklus

Leistungsanalyse zu verdeutlichen, wird folgend kurz auf das *Sampling* eingegangen. Dabei wird das zu messende Programm vom Laufzeitsystem in regelmäßigen Abständen unterbrochen und die Eckdaten des Prozesszustandes werden gespeichert. Durch das Sampling erfolgt eine regelmäßige Abtastung des Programms. Dies kann eine zeitliche Regelmäßigkeit mittels der Systemuhr bedeuten, aber auch ein regelmäßiger Interrupt aufgrund von Hardware-Metriken. Hardware-Metriken können dabei zum Beispiel Prozessordaten wie L1-Cache-Misses oder MFlop-Werte sein. Die regelmäßigen Ereignisse erlauben statistische Rückschlüsse auf das Laufzeitverhalten des Programms. Der größte Unterschied zur ereignisbasierten Leistungsanalyse besteht darin, dass die Aufzeichnung mittels externer Ereignisse arbeitet, wohingegen bei der ereignisbasierten Leistungsanalyse ein Ereignis innerhalb des Programms zur Leistungsmessung führt. Ein Vorteil dieser Messmethode ist ihre Einfachheit. Da die Informationen und Unterbrechungspunkte des Programms extern durch das Laufzeitsystem geregelt werden, ist keine Instrumentierung des Programms nötig.

Neben den Ereignissen, die als Symptom des eigentlichen Problems auftreten, ist es sinnvoll den Programmablauf aus einer lückenlosen Abfolge von Ereignissen zu modellieren. Durch eine lückenlose Abfolge von Ereignissen lassen sich Zustände des zu messenden Programms wohldefiniert behandeln. In solchen Daten können Informationen über jede Instanz eines Aufrufs gesammelt werden. Ferner lassen sich aus der Reihenfolge der Ereignisse genauere Rückschlüsse über das dynamische Verhalten ableiten. Um solch einen Ereignisstrom zu erzeugen, benötigt es allerdings einen stärkeren Eingriff in den Ablauf des Programms. Bei der ereignisbasierten Leistungsanalyse werden deshalb zusätzliche Instruktionen in den Programmcode eingeschleust, die das Anwenderprogramm selbst dazu veranlassen, an bestimmten Punkten Ereignisse zu speichern. Dieser Vorgang wird *Instrumentierung* genannt.

Die zusätzlichen Instruktionen zur Aufzeichnung können an verschiedenen Punkten in das Anwenderprogramm eingefügt werden. Man unterscheidet dabei zwischen *statischer* und *dynamischer Instrumentierung*. Wenn der Quelltext eines Programms vorliegt, kann dieser verändert werden, sodass nach erneutem Übersetzen die zusätzlichen Instruktionen mit ausgeführt werden. Die Änderung des Quelltextes kann dabei manuell oder automatisch durch ein zur Verfügung stehendes Werkzeug geschehen. Bei der automatischen Instrumentierung können unterstützende Funktionen des Compilers genutzt werden, die zusätzlichen Instruktionen während der Übersetzung einzubringen. Es ist auch die Bindung zu einer vorinstrumentierten Bibliothek möglich. Diese Verfahren verlangen dabei keine explizite Änderung des Quelltextes vor der Übersetzung. Die Wrapper der EPILOG-Bibliothek von KOJAK bilden eine solche vorinstrumentierte Bibliothek.

Wenn der Quelltext nicht zur Verfügung steht oder eine Neuübersetzung des Programms vermieden werden soll, kann auch ein Binärinstrumentierer eingesetzt werden. Dabei wird

dem bereits übersetzten Anwendungsprogramm auf binärer Ebene die Instrumentierung eingefügt. Bei den bisher vorgestellten Methoden der Instrumentierung sind die zusätzlichen Instruktionen fest im Programm verankert. Man spricht deshalb von *statischer Instrumentierung*. Bei der *dynamischen Instrumentierung* wird die Instrumentierung zur Laufzeit in das Programm eingebracht. Dadurch kann die Instrumentierung während der Laufzeit des Programms verändert und direkt auf das Verhalten des Programms abgestimmt werden. So können Instrumentierungen von unauffälligen Regionen zurück genommen werden, um die Störung des Programms auf ein Minimum zu beschränken. Da die Einschleusung von zusätzlichen Instruktionen zur Messung des Programms immer den Programmablauf verändert, ist es wichtig die zusätzlichen Instruktionen kompakt zu halten, um nicht, durch das Messsystem selbst, Leistungsprobleme in den Programmablauf zu induzieren.

Die im folgenden beschriebenen Methoden der ereignisbasierten Leistungsanalyse unterscheiden sich in ihrer Art, die gesammelten Daten auszuwerten. Beim *Profiling* werden die gesammelten Leistungsdaten noch während der Programmausführung akkumuliert. Dadurch lässt sich der benötigte Speicherplatz während des Programmablaufs minimieren. Dieser ist nur abhängig von der Anzahl der Messobjekte und nicht von der Laufzeit des Programms. Selbst ereignisreiche Programmabläufe können so ohne zu große Störungen des zu messenden Programms analysiert werden. Der Nachteil der Erstellung eines Profils während der Laufzeit ist, dass Informationen über den dynamischen Verlauf eines Programms verloren gehen. Dieser Informationsverlust tritt immer bei der Reduktion von Analysedaten auf. Zum Beispiel enthält ein Profil oft die Ausführungszeiten der einzelnen Funktionen des Programms. Diese können exklusive bzw. inklusive der Ausführungszeiten anderer Funktionen oder als absolute oder prozentuale Werte dargestellt werden. Eine genaue Identifizierung der Aufrufinstanz mit der längsten Ausführungszeit kann daraus allerdings nicht mehr gewonnen werden. Profildaten sind bei der Erzeugung ausschließlich prozesslokal, da eine Abstimmung der Daten zur Laufzeit unter den Prozessen den eigentlichen Programmablauf zu sehr stören würde. Daraus bedingt sich der Verlust von Informationen über externe Beeinflussung des lokalen Prozesses. Es kann zum Beispiel nicht mehr nachvollzogen werden, ob die hohe Ausführungsdauer von Funktion f auf Prozess x lokale Gründe hat, oder ob sie durch Ereignisse auf Prozess y beeinflusst wurde.

Das *Tracing* versucht durch die Erzeugung einer ausführlichen Spur, Informationen über den Programmablauf zu speichern. Eine *Spur* (trace) bezeichnet dabei eine chronologische Abfolge von Ereignissen, die den dynamischen Ablauf eines Programms modelliert. Dadurch kommen neue Möglichkeiten hinzu, Leistungsprobleme zu beschreiben und zu erkennen. Allerdings müssen meist so viele Daten gespeichert werden, dass sich die Speicherung auf einem lokalen Medium nicht vermeiden lässt und die Analyse dieser Daten zu aufwendig ist, um sie während der Laufzeit zu tätigen. Die Konzentration während der Ausführung des Programms liegt vielmehr nur auf der möglichst effizienten Speicherung der Ereignisse für eine spätere Analyse. Dabei werden die aufgetretenen Ereignisse zusätzlich zu ihren Attributen chronologisch gespeichert. Spuren können bei ungünstigen Programmabläufen leicht eine unhandliche Größe erreichen, sei es durch das Auftreten von sehr vielen Ereignissen über einen kurzen Zeitraum oder durch eine generelle lange Laufzeit. Ein Nachteil bei der Spur-Erzeugung ist, dass ihre Größe schlecht vorhergesagt werden kann. Der Anwender kann somit schlecht einen festen Speicherbereich definieren, der garantiert die gesamte Spur aufnehmen kann. Selbst bei einer möglichen vorherigen Speicherreservierung steht dem Anwendungsprogramm dieser Speicher nicht mehr zur Verfügung. Es wird somit nur ein kleiner Speicherbereich für die Spur reserviert, in dem Ereignisse gesammelt werden bis der Puffer voll ist und die tatsächliche Speicherung nicht weiter verzögert werden kann. Tracing eignet sich somit nur für Programme, die relativ kleine Spuren erzeugen, oder für die Analyse von Teilen von Programmabläufen.

Viele Leistungsprobleme ergeben sich durch die Flexibilität und die Komplexität der benutzten Programmierungskonzepte. Diese Komplexität kann meist nur noch durch umfangreiche Informationen über den globalen dynamischen Programmablauf modelliert werden, wodurch das Tracing eine verbreitete Methode darstellt. Genau wie moderne Hochleistungsrechner eine hybride Architektur besitzen, benutzen moderne Werkzeuge nicht nur eine einzige Methode der ereignisbasierten Leistungsanalyse, sondern kombinieren diese, um eine möglichst abgestimmte Analyse zu ermöglichen. Die Grenzen zwischen den einzelnen Methoden werden dadurch verwischt.

Instrumentierung durch Wrapper

Ein Grundstein bei der Identifizierung von Leistungsmerkmalen ist das Wissen darüber, zu welchem Zeitpunkt das Anwendungsprogramm welchen Programmteil ausführt. Dazu werden Eintritt und Austritt aus interessanten Regionen als Ereignis modelliert. Regionen sind dabei eine oder mehrere Instruktionen, die durch Einfügen einer Instrumentierung gemessen werden sollen. Funktionen werden für die Leistungsanalyse oft instrumentiert, da sie innerhalb eines Programms bestimmte Funktionalitäten kapseln. Um eine einfache Instrumentierung von Funktionen zu ermöglichen, werden *Wrapper* um die zu messenden Funktionen gelegt. Die Funktionen werden so durch die Messroutinen gekapselt. Dabei ist die Kernaufgabe, ein `ENTER`-Ereignis beim Betreten der Funktion und ein `EXIT`-Ereignis beim Austritt aus der Funktion zu erzeugen. Außerdem können bei den Ereignissen noch zusätzliche Attribute gespeichert werden, die weitere relevante Werte enthalten. Bei der Instrumentierung einer Funktion könnten dies die übergebenen Aufrufparameter sein.

Um keine Fehler durch den Analysevorgang in ein Anwendungsprogramm einzubringen, wird versucht, die vor der Leistungsanalyse nötigen manuellen Änderungen an einem Programm zu minimieren. Dabei gibt es verschiedene Vorgehensweisen, die dies ermöglichen. Um die Syntax der zu messenden Funktionsaufrufe nicht ändern zu müssen, ist es wünschenswert, die instrumentierte Funktion mit dem selben Namen der bisher nicht instrumentierten Funktion aufrufen zu können. Dazu kann man folgende Eigenschaft eines Linkers nutzen. In einem Programm müssen Symbole für Funktionsnamen eindeutig sein. Im Normalfall werden Funktionsnamen mit dem Prototyp auf globalem Geltungsbereich fest definiert. Eine weitere Funktion mit fester Bindung an dieses Symbol würde im Linkprozess einen Fehler verursachen. Eine schwache Bindung zu einem Funktionssymbol kann von einer anderen Bindung überschrieben werden. Bei mehreren schwachen Bindungen wird der Linker selbst auswählen, welche Funktion er mit dem Symbol verknüpft. Bei einer Bibliothek kann man auf diese Weise die Erstellung von Wrappern für die Bibliotheksfunktionen erheblich erleichtern. Eine Bibliotheksfunktion definiert neben einer globalen Symbolbindung auch noch eine schwache Bindung unter anderen Namen. Die schwache Bindung repräsentiert dabei den, für Anwenderaufrufe vorgesehenen, Funktionsnamen. Ein Wrapper für solch eine Bibliotheksfunktion kann nun durch den eigenen Prototypen eine feste Bindung für das Funktionssymbol erzeugen, das die schwache Bindung der eigentlichen Bibliothek überschreibt. Um die Bibliotheksfunktion weiterhin nutzen zu können, kann diese im Wrapper über das zweite Funktionssymbol aufgerufen werden. Eine auf diese Weise zwischengeschaltete Bibliothek ermöglicht es, Instrumentierungsanweisungen für ein Anwendungsprogramm einzufügen, ohne das Anwendungsprogramm selbst im Quelltext zu verändern. Bei Systemen, die diese Mehrfachbindung von Symbolen nicht unterstützen, kann das beschriebene Prinzip durch getrennte Bibliotheken für die zwei Symbole ermöglicht werden. Eine Angabe beider Bibliotheken und der zwischenzuschaltenden Bibliothek während des Link-Vorgangs ist hierzu nötig. Abbildung 2.2 stellt die Verwendung einer zwischengeschalteten Bibliothek schematisch dar. Das Anwendungsprogramm muss zur Leistungsanalyse nur neu gelinkt werden.

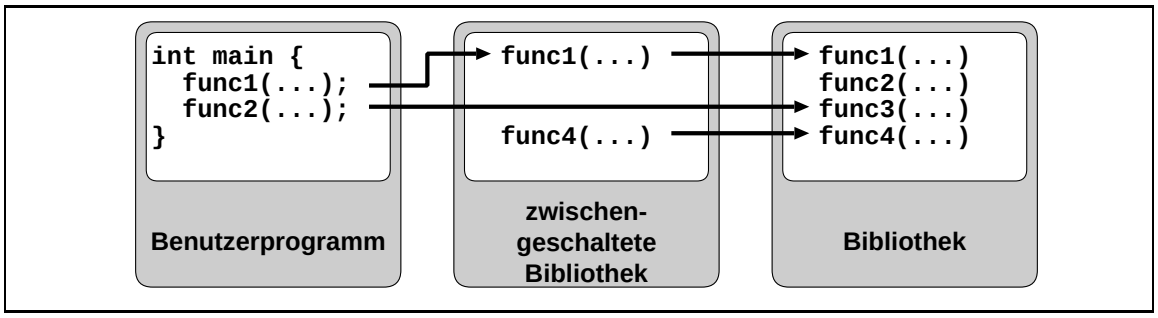


Abbildung 2.2: Ausnutzung einer schwachen Symbolbindung

2.2.2 Analyse der Ereignisdaten und Präsentation der Ergebnisse

Um die Leistung eines Programms automatisch bewerten zu können, muss definiert werden, welches Verhalten eines Programms überhaupt als negativ angesehen wird und welches nicht. Dazu werden zuerst *Leistungsmerkmale* (performance properties) definiert, die bewertungsfreie Ereignismuster darstellen, die Einfluss auf die Leistung eines Programms nehmen. Falls ein Leistungsmerkmal negative Auswirkungen auf die Leistung des Programms hat, so spricht man von einem *Leistungsproblem* (performance problem). Leistungsprobleme können in ihrem Auftreten verschieden stark auf die Leistung des Programms Einfluss nehmen. Alle gefundenen Leistungsprobleme werden vom Analysesystem automatisch bewertet und gewichtet. Das mit dem stärksten Gewicht wird als *Leistungsengpass* (bottleneck) des Programms bezeichnet. Die Vermeidung dieses Leistungsengpasses ergibt die größte Leistungsverbesserung.

Gerade bei verteilten Anwendungen liegen Leistungsprobleme oft in der Interaktion der einzelnen Prozesse. Oft können diese Leistungsprobleme nur durch eine globale Sicht auf den gesamten Ereignisstrom erkannt werden. Um diese globale Sicht bereitzustellen, müssen die prozesslokalen Ereignisströme zusammengeführt werden. In dieser globalen Sicht kann dann effektiv nach Leistungsmerkmalen gesucht werden. Um eine globale Sicht über den dynamischen Ablauf eines verteilten Programms zu erlangen, müssen die prozesslokalen Spuren zu einer Spur für das Programm vereint werden. Dabei ist es wichtig, redundante Informationen zu vermeiden, um die Größe der Spur zu minimieren. Gleichzeitig entsteht das Problem, dass Ereignisse in der Spur chronologisch sortiert auftreten müssen. In verteilten Systemen ist die Synchronisierung eines gesamten Systems ohne Hardwaresynchronisation sehr schwierig, da die einzelnen Werte der Systemuhren oft eine Verschiebung zueinander aufweisen. Gleichzeitig besitzen die verschiedenen Systemuhren verschieden starke Laufgeschwindigkeiten (drifts), die sich während der Laufzeit temperaturabhängig ändern können. Eine Synchronisation der Uhren mit angenommenem linearen Drift gibt keine absolute Sicherheit über die korrekte Reihenfolge der Ereignisse. In der Praxis ist aber der genaue Zeitpunkt eines Ereignisses nicht von unbedingter Wichtigkeit, solange die Ereignisse in einer logisch korrekten Reihenfolge in der Spur auftreten. Rolf Rabenseifner hat dieses Problem in seiner Dissertation adressiert [15]. Nachdem die einzelnen Spuren zu einer globalen Spur zusammengefügt wurden, können auf dem gesamten Strom Ereignisse gesucht werden.

Ereignismuster können einzelne Ereignisse sein oder durch Beziehungen zwischen mehreren Ereignissen definiert werden. Felix Wolf hat in seiner Diplomarbeit [17] eine Schnittstelle entworfen, die einen einfachen Zugriff auf Ereignisse und Programmzustände in Spuren ermöglicht. Dadurch wird es erheblich erleichtert, Leistungsmerkmale in einem Format zu beschreiben, das später eine einfache Suche ermöglicht. Auf diese Bibliothek wird in späteren Kapiteln noch weiter eingegangen, da sie ein Grundbestandteil der KOJAK-Werkzeugumgebung ist.

Wenn Ereignismuster gefunden und einzelnen Leistungsmerkmalen zugeordnet werden konnten, ist es nötig, diese dem Benutzer zu präsentieren. Dabei ist die Art der Visua-

lisierung ausschlaggebend. Die Analysedaten sollten in einer baumartigen Hierarchie von der Wurzel zu den Blättern immer detaillierter werden. Dies bedeutet nicht, dass die Informationen in Form eines Baums präsentiert werden sollten, sondern, dass für den Anwender das Problem oder die Probleme, auf den ersten Blick ersichtlich werden. Dann sollte der Anwender von dort immer detaillierter an die Probleme herangeführt werden. An der Wurzel sollte ein möglichst zusammenfassender Eindruck über die globale Leistung des Programms gewonnen werden können. Zu viele detaillierte Daten können hierbei die Sicht auf die eigentlichen Leistungsprobleme verschleiern. Das in Kapitel 2.3.3 beschriebene Visualisierungswerkzeug CUBE nutzt einen solchen abgestuften Detaillierungsgrad bei der Darstellung der Analysedaten.

2.2.3 Optimierung der Anwendung

Es ist sinnvoll, Leistungsprobleme nacheinander zu beseitigen, indem der Leistungsanalysezyklus mehrfach durchlaufen wird. So können geringere Leistungsprobleme, die eventuell erst durch den Leistungsengpass entstehen, erkannt werden. Unnötige Veränderungen des Programms zur vermeintlichen Optimierung werden verhindert. Nach der Optimierungsphase kann das teiloptimierte Programm, im Vergleich zu einer vorherigen Leistungsmessung, ein komplett neues Verhalten zeigen.

Optimierungen im Quellcode können oft nur manuell durch den Anwender selbst getätigt werden. Für einfache Leistungsprobleme können hierfür Optimierungshilfen angeboten werden. Aus Gesichtspunkten des Software-Engineering sind Programmiersprachen im Hochleistungsrechnen, wie Fortran und C, zu weit von dem Anwendungsproblem, also der Frage „Was soll das Programm leisten?“, entfernt, um automatische Optimierungen zu ermöglichen. Manche Optimierungen können nur durch die Wahl eines neuen Algorithmus getätigt werden. Beispielsweise existieren zwischen der seriellen und parallelen Datenverarbeitung zur optimalen Lösung eines Problems oft grundlegend andere Vorgehensweisen. Beispiele hierfür finden sich insbesondere bei Such- oder Sortieralgorithmen und der Anwendung mathematischer Verfahren. Diese neuen Ansätze müssen dann vom Anwender eingebracht werden.

2.3 Ereignisbasierte Leistungsanalyse mit KOJAK

Die in einer Kooperation zwischen dem Forschungszentrum Jülich und der Universität von Tennessee in Knoxville, USA, entwickelte Werkzeugumgebung KOJAK [7, 20], für die ereignisbasierte Leistungsanalyse von parallelen Programmen besteht aus mehreren Teilen. Die einzelnen Bestandteile erfüllen verschiedene Aufgaben innerhalb des Leistungsanalysezyklus.

Die Architektur der KOJAK-Werkzeugumgebung ist nah am Leistungsanalysezyklus entworfen. In der Instrumentierungsphase ermöglichen OPARI und EPILOG eine halbautomatische Instrumentierung der leistungsrelevanten Programmteile. compilerabhängig wird auch eine vollautomatische Instrumentierung unterstützt, doch nicht auf allen Plattformen sind die benötigten Compileroptionen verfügbar. EARL bietet während der Analysephase eine abstrakte Zugriffsschnittstelle auf den Ereignisstrom, sodass die Suche nach Ereignismustern durch EXPERT erleichtert wird. Die gesammelten Analyseergebnisse werden schließlich durch CUBE präsentiert. Abbildung 2.3 stellt ein Schema der Aufrufreihenfolge der einzelnen Werkzeuge bei der Leistungsanalyse dar.

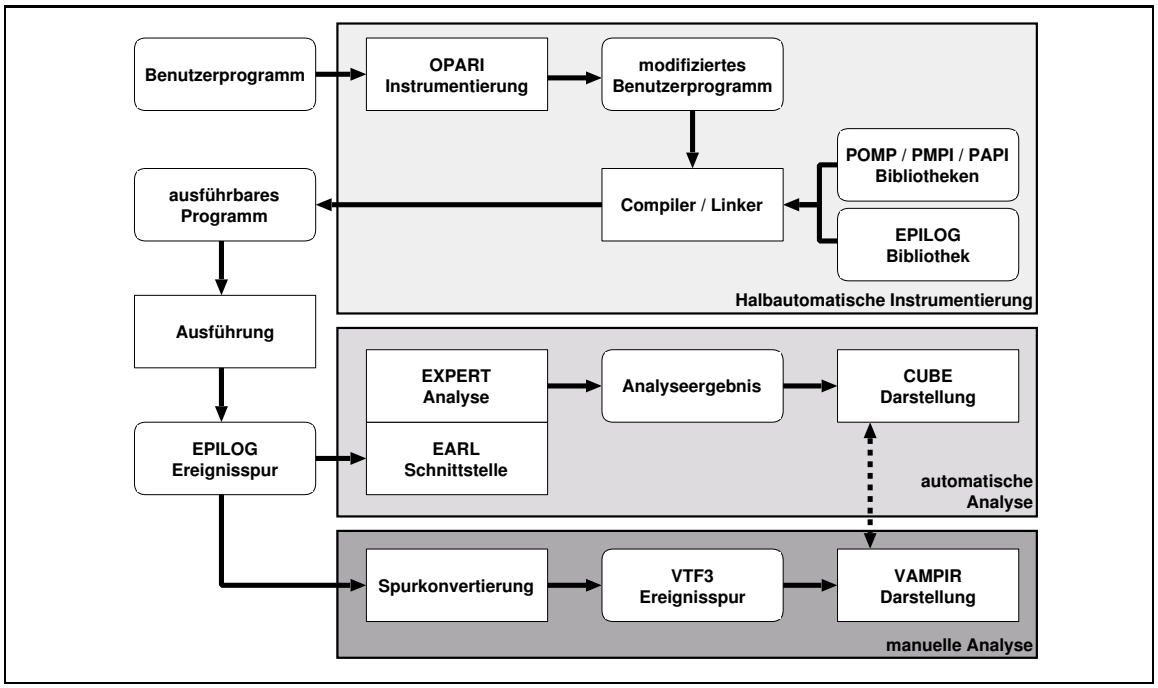


Abbildung 2.3: KOJAK-Architektur und die Schnittstelle zu VAMPIR

2.3.1 Statische Instrumentierung mit OPARI und EPILOG

OPARI

OPARI ist ein Quellcode-Transformationswerkzeug für die Instrumentierung von OpenMP. D.h. zur Instrumentierung von OpenMP-Strukturen muss der Quelltext des zu instrumentierenden Programms vorliegen, welcher vor der Compilierung verändert wird. OpenMP-Anweisungen werden im Quelltext durch Direktiven angegeben. Diese Anweisungen beziehen sich auf den der Anweisung folgenden Programmteil, welcher zur Übersetzungszeit entsprechend verändert wird. Um diese Anweisungen zu instrumentieren, wird von OPARI der Quelltext vor der Übersetzung nach OpenMP-Anweisungen durchsucht, und vor bzw. hinter den Anweisungen werden zusätzliche Instruktionen eingefügt. Der veränderte Quelltext wird wieder gespeichert. Die eingefügten Instruktionen basieren auf der POMP Bibliothek, welche die zeitliche Messung von OpenMP-Strukturen in C, C++ und Fortran Programmen ermöglicht.

EPILOG

MPI hat im Vergleich zu OpenMP eine andere Art der Einbindung. MPI Bibliotheken stellen eine Vielzahl von Funktionen zur Verfügung, die vom Benutzer im Programm aufgerufen werden können. Dadurch bietet sich die Möglichkeit an, diese Funktionen vorzuinstrumentieren. Dazu kann das bereits in Kapitel 2.2.1 auf Seite 9 beschriebene Verfahren zur Nutzung der schwachen Symbolbindung genutzt werden.

Der MPI-Standard definiert für jede Funktion zwei Symbole. Diese sind unter `MPI_Funktion` und `PMPI_Funktion` verfügbar. Wie diese beiden Symbole dem Anwender zur Verfügung gestellt werden, ist implementationsabhängig und MPI verwendet im allgemeinen eins der bereits beschriebenen Verfahren. Eine Bibliothek, die vorinstrumentierte Funktionen in Form einer Wrapper-Bibliothek bereitstellen will, kann diese unter dem Prototypen von `MPI_Funktion` zur Verfügung stellen. Innerhalb des Wrappers kann dann die Funktionalität der MPI Bibliothek über die Symbole `PMPI_Funktion` genutzt werden. Es ist wichtig, dass innerhalb eines solchen Wrappers nur die Symbole der PMPI-Schnittstelle genutzt

werden, um endlose Instrumentierungsschleifen zu verhindern. Da die `MPI_*`-Symbole auf diese Weise instrumentiert sind, braucht der Anwender sein zu analysierendes Programm nicht zu verändern, sondern nur mit den Bibliotheken neu zu linken.

Durch die Kapselung der Funktionen entstehen allerdings Leistungseinbußen. Deshalb werden nicht alle MPI-Funktionen instrumentiert, sondern nur die, die für eine spätere Analyse benötigt werden. Darunter fallen alle Funktionen zur Punkt-zu-Punkt-Kommunikation und kollektiven Kommunikation sowie, im Rahmen dieser Arbeit, auch der einseitigen Kommunikation, außerdem noch alle Funktionen zur Kommunikator- und Gruppengenerierung, sowie die bei der einseitigen Kommunikation anfallenden Synchronisationsfunktionen. Die Instrumentierung dieser Hilfsfunktionen ist wichtig, da sie für die Analyse notwendige Daten aufzeichnen.

Eine Instrumentierung aller MPI-Funktionen ist nur sinnvoll, wenn der Benutzer in einem konkreten Fall diese umfassenden Informationen benötigt, z.B. zur Visualisierung der gesamten Programmabläufe. Die Messung einer Funktion beeinträchtigt immer die Leistung des zu messenden Programms. Für die automatische Leistungsanalyse sollten somit übermäßige Störungen im Programmablauf vermieden werden.

EPILOG enthält, neben den gerade beschriebenen PMPI-Wrappern und den bereits erwähnten POMP-Funktionen, Funktionen zur Ereigniserfassung und -speicherung, sowie Routinen zur Zeitsynchronisation und Aufzeichnung von Hardwarecounter-Daten mittels der PAPI-Bibliothek [16].

EPILOG speichert alle während des Programmablaufs auftretenden Ereignisse im EPILOG-Spurformat [21]. In diesem Format werden die Ereignisdaten in einem binären Format abgelegt. Es wird grundsätzlich zwischen Definitionseinträgen und Ereigniseinträgen unterschieden. Ereigniseinträge beschreiben den dynamischen Ablauf des gemessenen Programms. Definitionseinträge stellen für die Ereignisse Objekte zur Verfügung, auf die diese sich beziehen können, um redundante Informationen in der Ereignisspur zu vermeiden. Das bedeutet, dass ein Kommunikator einmal definiert wird und Ereignisse diesen Kommunikator referenzieren, ohne alle damit verbundenen Daten erneut speichern zu müssen. Definitionseinträge sind dabei global. D.h. bei der Zusammenführung der prozesslokalen Spuren zu einer globalen Spur werden doppelte Definitionen gefiltert und Ereignisse so angepasst, dass sie das richtige Objekt referenzieren.

2.3.2 Analyse des Ereignisstroms mit EARL und EXPERT

EARL

EARL [19] bietet eine abstrakte Schnittstelle zur Analyse und Verarbeitung des EPILOG-Spurformats. Der Zugriff auf ein beliebiges Ereignis im Ereignisstrom wird ermöglicht und zusätzlich kann der Ausführungszustand des Programms an diesem Ereignis zurückgeliefert werden. EARL verknüpft zusammengehörige Ereignisse innerhalb des Ereignisstroms. So können auf einfache Weise Ereignismuster im Ereignisstrom analysiert werden.

Die Hauptaufgabe von EARL liegt in der Vereinfachung der Definition von Mustern im Ereignisstrom. Es bietet somit die Basis für die spätere Analyse. EARL ist in C++ implementiert und bietet Schnittstellen für Python und C++. Die Python Schnittstelle ermöglicht dem Entwickler, die von EARL bereitgestellten Funktionen interaktiv zu nutzen und schnell Prototypen von Leistungsmerkmalen zu erzeugen.

Bei der Definition der Ereignistypen wird ein objektorientiertes Modell genutzt. Durch Vererbung werden redundante Informationen vermieden und Zusammengehörigkeiten auf Definitionsebene ausgedrückt. Die Abbildung 2.4 zeigt eine an UML angelehnte Darstellung der Klassenhierarchie, wie sie in EARL 2.0 definiert ist. Anhand der Köpfe der dargestellten

Ereignistypen ist erkennbar, ob es sich um einen OpenMP-spezifischen, MPI-spezifischen oder einen allgemeinen Ereignistyp handelt. Zur besseren Übersichtlichkeit sind nur die Attribute und nicht die Methoden der einzelnen Klassen dargestellt.

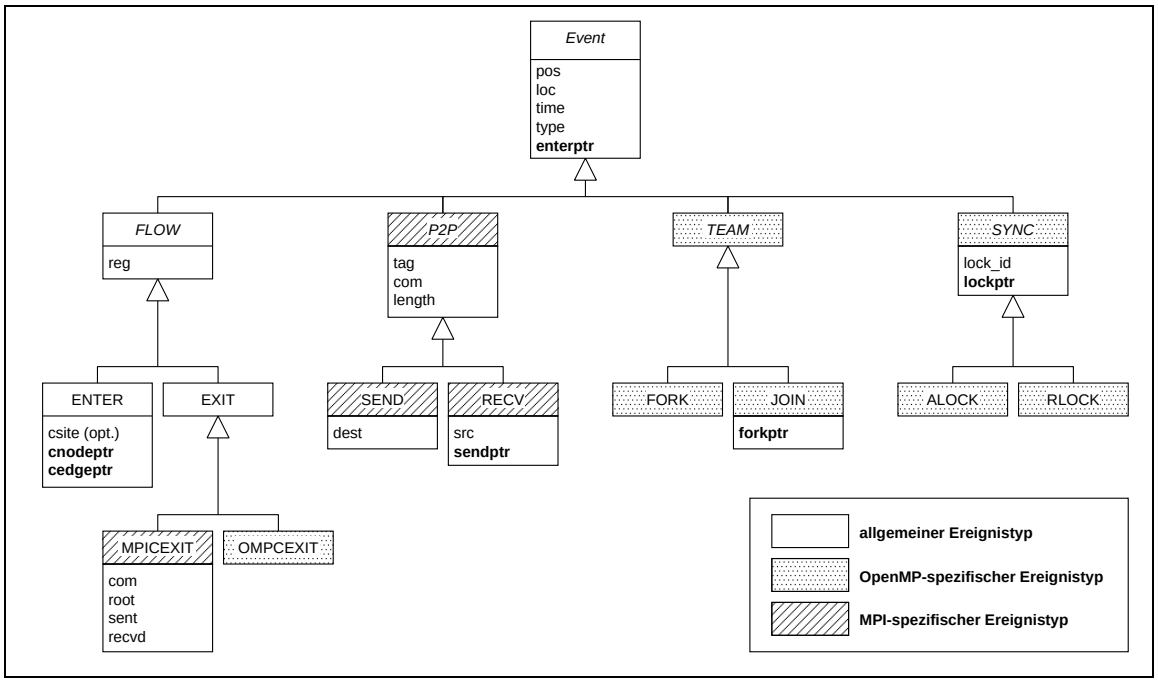


Abbildung 2.4: Ereignistypen in EARL 2.0

Event stellt die Wurzelklasse aller Ereignisse des Ereignisstroms dar. Ein Ereignis wird beschrieben durch seine Position im Strom (pos), den Ort (loc) und den Zeitpunkt (time) seines Auftretens sowie seinen Typ (type). Von diesem Basisereignis leiten sich zunächst Zwischenklassen ab, die als Elternklassen für die im Ereignisstrom auftretenden Ereignisse dienen. Die Wurzelklasse und alle Ereignisklassen, die direkt von ihr erben, sind abstrakt, d.h. sie können nicht im Datenstrom auftreten. Sie dienen dazu die abgeleiteten Klassen gemäß ihrer Zugehörigkeit zusammenzufassen und gemeinsame Attribute zu definieren. FLOW ist Elternklasse aller Kontrollfluss-Ereignisklassen. Das Attribut reg bezeichnet die Region, die betreten bzw. verlassen wird. ENTER.csite ist ein optionales Attribut, welches bei verwendeter Callsite-Instrumentierung die Callsite-Identifikation speichert. Bei den Austrittsereignissen EXIT, OMPCEXIT und MPICEXIT besitzt nur das letztere zusätzliche Attribute, die die Parameter einer kollektiven Kommunikation speichern. Dies sind der Kommunikator (com), die Wurzel (root) der Operation sowie die Länge der gesendeten (sent) und empfangenen (recvd) Nachrichten in Bytes. Durch Unterklassen von P2P werden Datentransfers der Punkt-zu-Punkt-Kommunikation modelliert Im Attribut tag wird das Tag einer Punkt-zu-Punkt-Kommunikation gespeichert und com gibt den Bezeichner für den genutzten Kommunikator wieder. Durch length wird die Länge der übertragenen Nachricht in Bytes gespeichert. Die Quelle des Punkt-zu-Punkt-Datentransfers ist im Attribut src des Ereignistyps RECV gespeichert, um den Ort des zugehörigen SEND-Ereignisses identifizieren zu können. In gleichem Maße wird im Attribut dest des Ereignistyps SEND der Ort des zugehörigen RECV-Ereignisses gespeichert. In TEAM werden Anfangs- (FORK) und Endereignisse (JOIN) eines fork-join-basierten Thread-Modells zusammengefasst. SYNC dient schließlich als Elternklasse für die im Thread-Modell verwendeten Sperren ALOCK (acquire lock) und RLOCK (release lock). Durch das Attribut lockid wird dabei eine Sperre eindeutig identifiziert.

Um Abläufe durch Ereignisse besser darstellen zu können, wird ein Zustand ebenfalls als endliche indizierte Menge von Ereignissen modelliert und bildet eine Teilmenge des gesamten Ereignisstroms. Ein Zustand S_i geht durch das Auftreten des Ereignisses E_i in den

nächsten Zustand S_{i+1} über. Daraus ergibt sich der dynamische Ablauf des Programms als Sequenz solcher Zustände. Diese Zustandssequenz kann formal als $S = \{S_0, \ldots, S_i, \ldots, S_n\}$ beschrieben werden, wobei n die Anzahl der bisher eingetretenen Ereignisse ist, und S_0 den Anfangszustand mit leerer Ereignismenge beschreibt. Dadurch beschreibt ein Zustand die gesamte *Ereignisgeschichte* (event history) beschrieben.

Um Verbindungen zwischen den einzelnen Ereignissen beschreiben zu können, werden *Zeigerattribute* (pointer attributes) definiert. Diese Zeiger referenzieren ein anderes bereits geschehenes Ereignis. In der Abbildung 2.4 sind diese Zeigerattribute durch Fettdruck gekennzeichnet. Sie zeigen im Ereignisstrom also immer zurück. Um Situationen beschreiben zu können, in denen kein sinnvolles Ereignis referenziert werden kann, ist es erlaubt diese Zeigerattribute mit *null* zu belegen. RECV.*sendptr* enthält einen Zeiger auf das zugehörige SEND-Ereignis. Analog enthält JOIN.*forkptr* einen Zeiger auf das zugehörige FORK-Ereignis. Event.*enterptr* referenziert das ENTER-Ereignis, der zuletzt betretenen Region. Gibt es keine übergeordnete Region, die referenziert werden kann, so ist dies die erste im Programmablauf betretene Region, das Hauptprogramm selbst. Anhand dieses Zeigers kann zum angegebenen Ereignis schnell der Aufrufpfad erstellt werden. SYNC.*lockptr* ermöglicht die Referenzierung des letzten Ereignisses vom Type SYNC mit der gleichen SYNC.*lockid*. Auf diese Weise wird eine Liste aller für diese Sperre behandelten Zugriffssperren erstellt.

EXPERT

EXPERT nutzt die von EARL bereitgestellte Schnittstelle zur Definition von Ereignismustern. Zur Leistungsanalyse werden nur solche Ereignismuster definiert, die Einfluss auf die Leistung des Programms haben können. Diese werden Leistungsmerkmale genannt. EXPERT ist in C++implementiert und benutzt objekt-orientierte Konzepte zur Implementierung von Leistungsmerkmalen. Alle Leistungsmerkmale leiten sich deshalb von der Basisklasse Pattern ab. Sie stellt das Skelett eines jeden Leistungsmerkmals dar. Pattern selbst erbt von der abstrakten Klasse Callback alle benötigten Callback-Funktionen für die definierten Ereignisse. Eine konkrete Merkmalsklasse kann dann die Ereignis-Callbacks überschreiben, die sie zur Erkennung des dazugehörigen Musters benötigt.

Abbildung 2.5 gibt die Leistungsmerkmale und Leistungsprobleme wieder, die in EXPERT 3.0 definiert sind. Weiße Felder geben dabei Leistungsmerkmale wieder, für die während der Analysephase Profildaten gesammelt werden. Innerhalb der Hierarchie der Leistungsmerkmale können diese Daten inklusiv oder exklusiv angegeben werden. Eine inklusive Angabe der Informationen beinhaltet alle Zeiten, inklusive der von diesem Merkmal abgeleiteten Merkmale. Eine exklusive Betrachtung gibt Informationen über das Leistungsmerkmal ohne die unterliegenden Leistungsmerkmale an. Die Werte, die innerhalb dieser Hierarchie gemessen wurden, sind erst einmal ohne Wertung angegeben. Wenn ein Großteil der Zeit in Execution verbracht wird, heißt dies lediglich, dass das Programm größtenteils eigene Berechnungen ausgeführt hat. Wenn der Anteil von MPI oder OpenMP größer ist, kann dies darauf hindeuten, dass anteilig zuviel Zeit in Funktionen dieser Gruppen verbracht wurde.

Floating Point Inst. und L1 Cache Miss geben Leistungsmerkmale wieder, die auf Hardwaremetriken beruhen. Cache Misses sind in ihrer Natur nicht zu vermeiden, und so muss der Anwender selbst entscheiden, ob die Zahl der Cache Misses im Programm optimal ist. Gleiches gilt auch für die Bewertung der Informationen über die erreichten Fließkommaoperationen innerhalb einer Funktion. Innerhalb einer Kommunikationsfunktion werden diese vergleichsweise gering ausfallen, wo hingegen sie bei einer Benutzerfunktion zur Berechnung eines Ergebnisses möglichst hoch sein sollten.

Graue Boxen bezeichnen Leistungsprobleme, die nur durch im Ereignisstrom auftretende Muster bestimmt werden können. Die Suche dieser Muster wird durch die von EARL be-

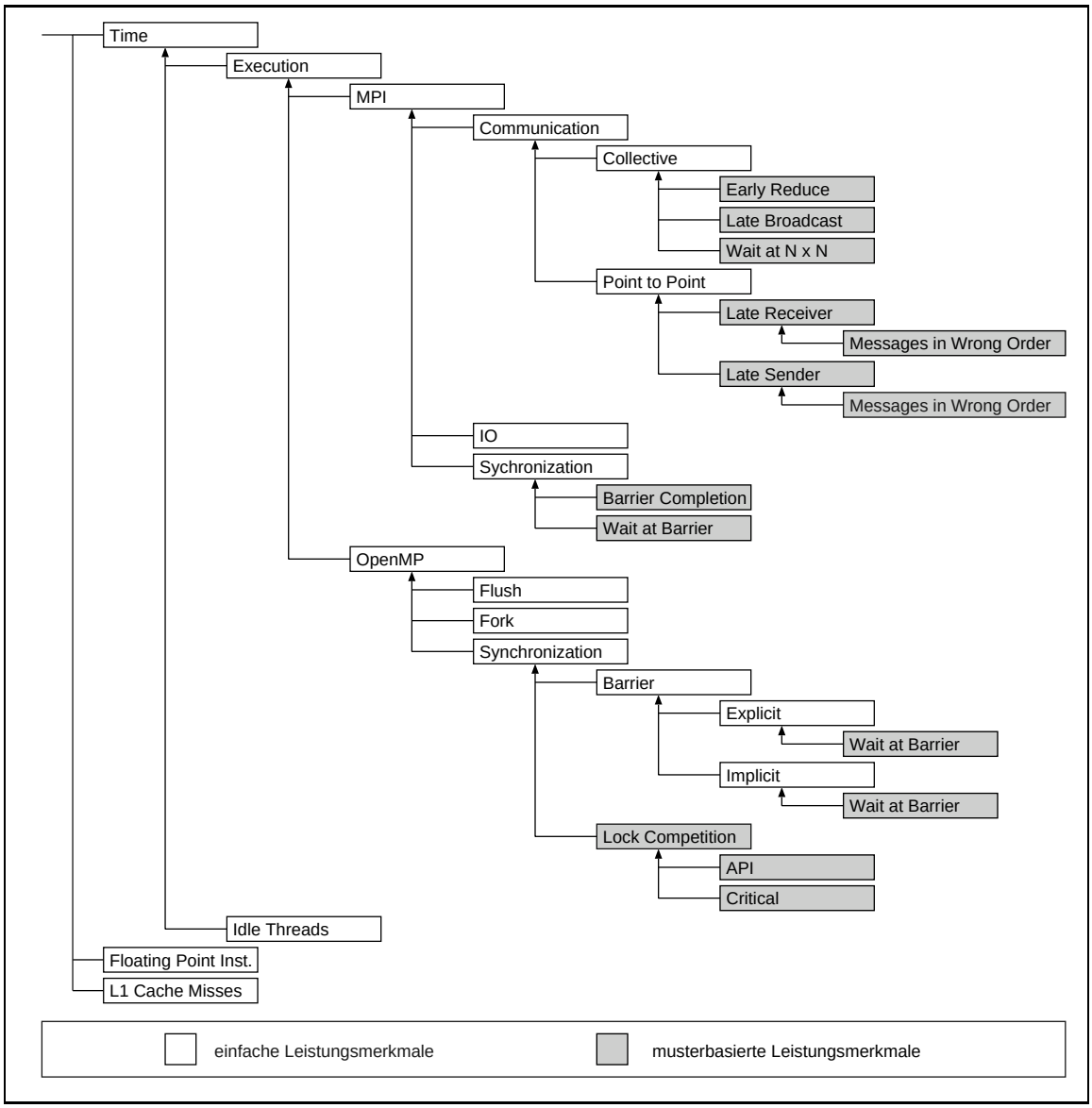


Abbildung 2.5: Leistungsmerkmale und Leistungsprobleme in EXPERT 3.0

reitgestellten Beziehungen der Ereignisse untereinander im Ereignisstrom erleichtert. Als Gewichtung wird ihnen die Zeit zugewiesen, die durch dieses Merkmal verloren wird, z.B. aufgetretene Wartezeiten. Die Gewichtung eines Leistungsmerkmals ist ein Teil der Gewichtung des hierarchisch übergeordneten Leistungsmerkmals. Auf diese Weise ist in Time die gesamte Ausführungszeit akkumuliert. Von dort werden entsprechende Anteile der Gewichtung eines Knotens den Zweigen zugeordnet. Eine Gewichtung muss dabei nicht zu 100% auf die weiterführenden Zweige verteilt sein. Der Rest der Gewichtung bleibt dann dem Knoten selbst zugeordnet. Knoten können in ihrer damit nicht höher sein als die Gewichtung des übergeordneten Knotens. Auf diese Weise kann der Anwender sehr einfach Fragen wie z.B. „Wieviel Zeit nimmt das Leistungsproblem Early Reduce von der gesamten kollektiven Kommunikationszeit in Anspruch?“ beantworten. Solche Fragestellungen sind bei der Leistungsbewertung einer Anwendung wichtig, da sie unter Umständen Ansatzprobleme und nicht Implementationsprobleme aufzeigen. Wenn grundsätzlich mehr Zeit in der Kommunikation verbracht wird, als in der Berechnung des eigentlichen Problems, ist zu überlegen, ob der gewählte Ansatz nicht verworfen werden muss.

EXPERT berechnet die Gewichtung jedes Merkmals für jeden Punkt im Aufrufgraphen des Programms sowie für jeden am Programmablauf beteiligten Thread bzw. Prozess. Daraus

lässt sich in der geometrischen Anschauung ein Würfel bilden, bei dem die Kanten entsprechend durch die Leistungsmerkmale, die Regionen und die Prozesse bzw. Threads beschrieben werden. Daraus leitet sich der Name des Präsentationswerkzeugs dieser Analysedaten ab, CUBE.

2.3.3 Präsentation der Analyseergebnisse durch CUBE

CUBE arbeitet auf Daten, die durch EXPERT im XML Format zur Verfügung gestellt werden. Dabei ist CUBE so flexibel, dass es alle Definitionen der Leistungsmerkmale und ihre Hierarchie aus den eingelesenen Daten selbst entnimmt. Dies bedeutet zum einen, dass CUBE nicht an neue Leistungsmerkmale angepasst werden muss, um diese verarbeiten zu können, und zum anderen kann es auf diese Weise auch von anderen Analysewerkzeugen als Anzeigewerkzeug benutzt werden.

Die dreigeteilte Visualisierung der graphischen Benutzerschnittstelle, ermöglicht eine einfache und intuitive Visualisierung, relativ zu den ausgewählten Baumknoten. In allen drei Teilfenstern der Benutzerschnittstelle, die in Abbildung 2.6 dargestellt ist, werden die Daten hierarchisch als Baumstruktur dargestellt. So können durch Auf- und Einklappen der Baumknoten leicht Daten konsolidiert oder detailliert aufgeschlüsselt werden. Jeden Knoten erhält eine farbige Kodierung anhand seiner Gewichtung. Um den Blick des Anwenders nicht durch viele Farben abzulenken, werden alle Gewichtungen, die einen Schwellwert nicht überschreiten durch grau kodiert.

Die Hierarchie der Leistungsmerkmale aus Abbildung 2.5 ist im linken Teil des Fensters abgebildet. Je nachdem ob ein Zweig eingeklappt oder ausgeklappt ist, werden die inklusiven oder exklusiven Profildaten angezeigt. So kann der Anwender gezielt und dynamisch die darzustellenden Informationen beeinflussen. Durch die Selektion eines Leistungsmerkmals ergibt sich im mittleren Feld die Verteilung dieses Leistungsmerkmals auf die einzelnen

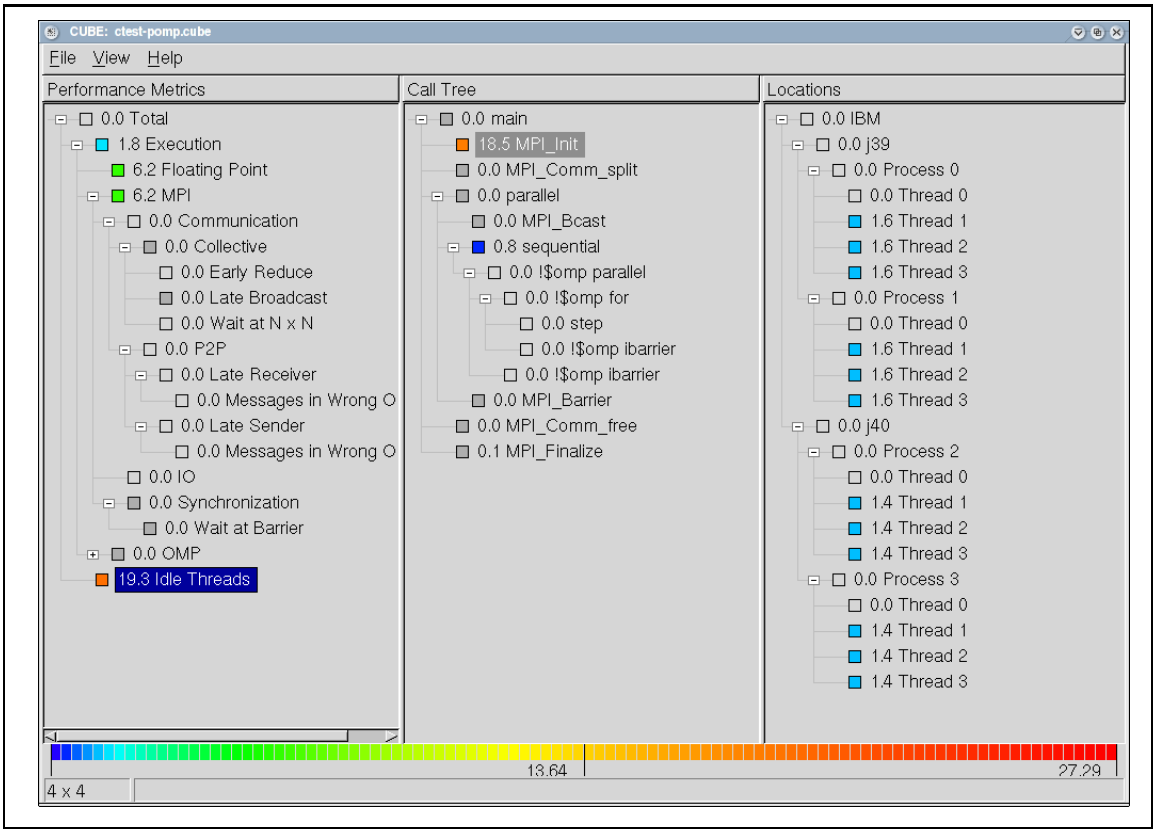


Abbildung 2.6: Dreigeteiltes Präsentationsfenster von CUBE

Regionen. Die Selektion einer einzelnen Region wiederum zeigt daraufhin im rechten Feld die Verteilung auf die einzelnen Prozesse. Auf diese Weise kann der Anwender schnell ein Ungleichgewicht im Kommunikationsverhalten identifizieren oder aufgetretene Leistungsprobleme schnell eingrenzen.

2.3.4 Darstellung des Ereignisstroms durch VAMPIR

KOJAK verfügt über keine eigene Komponente zur Visualisierung des Ereignisstrom. Allerdings ist es möglich einen durch EPILOG aufgezeichneten Ereignisstrom in das VTF3-Spurformat zu konvertieren, welches von VAMPIR 3 (und höher) verarbeitet werden kann. Mit Bereitstellung dieser Spurkonvertierung erhält der Anwender die Möglichkeit, neben der automatischen Analyse durch EXPERT auch eine manuelle Analyse des Ereignisstroms vorzunehmen. Dies kann sinnvoll sein, da EXPERT wissensbasiert arbeitet, und somit nur vorher definierte Ereignismuster suchen und erkennen kann. Ein erfahrener Anwender und Entwickler von parallelen Programmen kann darüber hinaus allerdings auch bisher unbekannte Leistungsprobleme identifizieren, indem er innerhalb der Diagramme ebenfalls Muster erfasst und manuell analysiert.

Die Zeitlinien-Diagramme von VAMPIR geben dem Anwender dazu die Möglichkeit, doch je umfangreicher der Ereignisstrom ist, und je mehr beteiligte Prozesse in der Ereignisspur enthalten sind, desto mehr gehen die Muster solcher bisher unbekannten Leistungsprobleme in der Masse der Daten unter.

Kapitel 3

MPI-2: Einseitige Kommunikation

3.1 Motivation

1995 verabschiedete das MPI Forum den Standard 1.1 [9] zur Kommunikation über Nachrichtenaustausch, das Message Passing Interface. Der MPI Standard beschreibt eine Sammlung von Funktionen, die zur Kommunikation und ihrer Organisation benötigt werden. Dadurch soll die Möglichkeit geschaffen werden, Anwenderprogramme plattformübergreifend und portabel zu entwickeln. Das Konzept des Nachrichtenaustauschs zeichnet sich dadurch aus, dass ein Prozess eine Nachricht aus einem Puffer an einen anderen Prozess schickt. Dieses Konzept ermöglicht die Kommunikation in einer Umgebung mit verteiltem Speicher.

In dieser ersten Version des MPI Standards wurden Funktionen zur Punkt-zu-Punkt und zur kollektiven Kommunikation unterstützt. An einem einzelnen Aufruf einer Punkt-zu-Punkt Kommunikation sind immer nur Sender und Empfänger beteiligt. Bei der kollektiven Kommunikation sind bei einem Aufruf meist mehrere Kommunikationspartner beteiligt, wobei die benötigten Einzelkommunikationen zwischen den Prozessen innerhalb des einen Aufrufs getätigt werden. Dies ermöglicht die Bereitstellung von Funktionen mit optimierten Kommunikationsmustern bei der Datenverteilung mit mehreren Prozessen.

Bereits vor der offiziellen Veröffentlichung von MPI 1.1 begannen im MPI Forum die Arbeiten an einer Erweiterung. 1997 wurde dann der Standard 1.2 veröffentlicht, welcher Berichtigungen zum bis dahin geltenden Standard enthielt. Zusätzlich wurden Erweiterungen von MPI 1.2 unter MPI 2.0 [10] veröffentlicht. Damit wurde auch ein weiteres Kommunikationskonzept unterstützt: die *einseitige Kommunikation* (one-sided communication). In manchen englischsprachigen Texten wird auch von single-sided communication gesprochen. Der wesentliche Unterschied zur zweiseitigen Kommunikation (send/receive) ist, dass nur *ein* Prozess die Parameter für beide Kommunikationspartner definiert. Die einseitige Kommunikation ist eine Möglichkeit des *entfernten Speicherzugriffs* (Remote Memory Access). Deshalb werden die Datentransfer-Operationen innerhalb dieser Arbeit auch als RMA-Operationen bezeichnet.

Die einseitige Kommunikation ermöglicht Kommunikationsabläufe, die mit der Punkt-zu-Punkt Kommunikation oder der kollektiven Kommunikation nicht effizient zu erreichen sind [11]. Dies ist auch der primäre Gesichtspunkt bei der Definition des Standards gewesen. Es gibt Algorithmen, bei denen Prozesse zusätzliche Daten für eine Berechnung von anderen Prozessen erhalten müssen. Die schon in MPI 1.1 definierte Punkt-zu-Punkt Kommunikation und die kollektive Kommunikation kennen nur Funktionen, bei denen ein Prozess aktiv Daten schickt, und die Empfänger schon vorher fest stehen müssen. Die einseitige Kommunikation ermöglicht es nun auch, dass ein Prozess Daten selbst von einem Prozess beziehen kann, ohne dass dieser explizit Daten senden muss. Damit ist es mög-

lich in bestimmten Fällen die Anzahl der Nachrichten von einer Komplexität von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n)$ zu reduzieren. Die RMA-Operationen bieten insgesamt ein vollständiges Kommunikationsparadigma, welches es ermöglicht alle Arten der Kommunikation mit Hilfe der einseitigen Kommunikation zu modellieren. Darüber hinaus wurden noch drei verschiedene Synchronisationsmechanismen entworfen auf die später noch tiefer eingegangen wird.

Schon vor der Erweiterung des MPI Standards gab es Möglichkeiten zum entfernten Speicherzugriff. Diese waren allerdings nicht standardisiert sondern herstellerspezifisch. Damit konnten sie speziell auf die zur Verfügung stehende Hardware angepasst werden. So erreichen diese Bibliotheken, wie z.B. SHMEM von Cray Inc., sehr gute Leistungseigenschaften. Die einseitige Kommunikation innerhalb des MPI-Standards ist nicht dazu konzipiert in diesem Bereich mit hardwarenäheren Bibliotheken konkurrieren zu können. Allerdings eröffnet sie einen standardisierten Zugang zu plattformübergreifender Funktionalität, die bisher in MPI nicht gegeben war.

Die Leistung des Datentransfers wird größtenteils durch die verschiedenen Synchronisationsmodelle beeinflusst. Komplexe und übergreifende Synchronisationsmuster bringen zusätzliche Latenz in die Datenübertragung. Vergleiche mit Daten von kollektiven Kommunikationsfunktionen zeigen daher auch eine ähnliche Leistung für die einseitige Kommunikation [5]. Luecke und Hu zeigen, dass die RMA-Operationen nicht einen generellen Vorteil gegenüber anderen Kommunikationskonzepten bieten, allerdings zeigen Bemühungen der NASA nach eigenen Angaben bei einer Umstellung eines Programms zur Atmosphären-Modellierung von nicht-blockierender Punkt-zu-Punkt Kommunikation auf einseitige Kommunikation eine Leistungssteigerung von 39 Prozent [13].

In den folgenden Unterkapiteln wird auf die genaue Funktionsweise der Datenübertragung und der Synchronisation dieser Datenübertragung innerhalb der einseitigen Kommunikation eingegangen. Damit werden die Grundlagen für die in Kapitel 4 entwickelten Ereignismodelle für die einseitige Kommunikation geschaffen.

3.2 Fenster als Rahmen für den Datenaustausch

Die einzelnen Prozesse können nicht auf beliebige Speicherbereiche des Kommunikationspartners schreiben oder lesen. Die Kommunikationspartner müssen vorher gemeinsam Speicherbereiche für den Datenaustausch definieren. Solche Speicherbereiche werden Speicherfenster genannt. In dieser Arbeit wird auch die kurze Form Fenster benutzt. Die Definition eines Fensters geschieht über die kollektive Operation `MPI_Win_create`, welche ein Handle zurückliefert, mit dem im späteren Verlauf des Programms die Zugriffe koordiniert werden. Dabei wird einem Fenster die Gruppe von Prozessen zugeordnet, die an der Definition beteiligt waren. Andere Prozesse können nicht nachträglich ebenfalls auf dieses Fenster zugreifen. Der Standard erlaubt jedoch, verschiedene Fenster auf überlappende Speicherbereiche zu definieren. Der Anwender muss hierbei allerdings darauf achten, die RMA-Zugriffe so zu koordinieren, dass keine gleichzeitigen Zugriffe auf die selben Speicherbereiche geschehen. Da die Definition eines Fensters kollektiv ist, müssen auch die Prozesse, die keinen Speicherbereich zur Verfügung stellen wollen, `MPI_Win_create` aufrufen. Um kein eigenes, lokales Fenster zu definieren, sondern nur das Handle für Zugriffe auf Speicherfenster anderer Prozesse zu erhalten, kann ein Prozess als Puffergröße 0 angeben.

Grundsätzlich können Fenster in beliebigen Teilen des Prozessspeichers liegen. Jedoch empfiehlt das MPI Forum dem Benutzer, diese Speicherbereiche durch die Funktion `MPI_Alloc_mem` anzufordern, da dies auf manchen Systemen zu besserer Leistung führt. Gleichzeitig erlauben sie den Herstellern, Zugriffe auf Fenster mit passiver Synchronisierung (siehe Kapitel 3.4.2) ausschließlich auf solche Speicherbereiche zu beschränken. Diese Einschränkung bildet einen der großen Kritikpunkte am MPI Standard, in Bezug auf seine

Eignung als Plattform zur Implementierung von GAS (global address space) Programmiersprachen [2].

Um ein Fenster wieder freizugeben, also das Handle zu löschen, wird dies der kollektiven Funktion `MPI_Win_free` übergeben. Die Variablen und Werte, die in dem Speicherbereich stehen, bleiben aber für den lokalen Prozess weiterhin bestehen. Um einen über `MPI_Alloc_mem` reservierten Speicherbereich freizugeben, ist die Funktion `MPI_Free_mem` nötig.

3.3 Datenübertragung

Um auf den definierten Fenstern Daten auszutauschen, stehen drei verschiedene Funktionen zur Verfügung:

`MPI_Get` erlaubt einem Prozess Daten aus einem Fenster in den eigenen lokalen Speicher zu laden.

`MPI_Put` erlaubt einem Prozess Daten in ein freigegebenes Fenster zu speichern.

`MPI_Accumulate` erlaubt darüber hinaus die Verknüpfung der gesendeten Daten mit einer Reduktionsoperation auf der Seite des Empfängers.

Durch eine Reduktionsoperation können von verschiedenen Prozessen gesendete Daten direkt mit einer Operation wie Summen- oder Produktbildung verknüpft werden. Dies hat den Vorteil, nicht zusätzlich Speicherbereiche zu benötigen, um die gesendeten Daten zwischenzuspeichern, bevor sie durch eine lokale Funktion verknüpft werden. Als Verknüpfung sind dabei alle von MPI vordefinierten Reduktionsoperationen erlaubt. Benutzerdefinierte Operationen, wie sie bei einem `MPI_Reduce` erlaubt sind, sind hier nicht zugelassen. Dafür ist die zusätzliche Operation `MPI_REPLACE` definiert, bei der die Daten im Fenster jeweils immer durch die geschickten Daten ersetzt werden. `MPI_Put` kann somit als Spezialfall eines `MPI_Accumulate`, in Verbindung mit der Operation `MPI_REPLACE`, gesehen werden.

Der aufrufende Prozess einer dieser drei Operationen wird Ursprungsprozess genannt. Der Prozess, der das Speicherfenster für den Zugriff öffnet, heißt Zielprozess.

Im Hinblick auf Leistungseigenschaften von parallelen Programmen sind blockierende Funktionen oft eine Stelle, an der Wartezeiten für Prozesse entstehen. Die Aufrufe der drei beschriebenen MPI-RMA-Operationen sind nicht blockierend. Die bedeutet, dass sie den Kontrollfluss an das Anwenderprogramm zurückgeben, bevor sie abgeschlossen sind. Der MPI Standard erfordert, zusätzlich zu einer RMA-Operation, die Synchronisation des benutzten Speicherfensters, um die RMA-Operation abzuschließen. Wartezeiten durch ungünstige Zugriffszeitpunkte können so vermieden werden, verschieben sich für die Analyse allerdings in die zugehörigen Synchronisationen.

Ein Zeitraum zwischen zwei Synchronisationsaufrufen wird *Zugriffsepoche* (access epoch) genannt. Falls in einer Zugriffsepoche eine RMA-Operation getätigt wird, ist die Epoche eine RMA-Zugriffsepoche, sonst eine lokale Zugriffsepoche. Auf die lokalen Variablen eines Fensters darf während einer RMA-Zugriffsepoche nicht durch lokale LOAD/STORE-Befehle zugegriffen werden, bevor nicht der zugehörige Synchronisationsaufruf beendet ist.

Um effiziente Implementierungen zu ermöglichen, sind gleichzeitige Zugriffe auf sich überlappende Sendepuffer erlaubt. Dadurch können möglichst viele RMA-Operationen mit einem Synchronisationsaufruf bearbeitet werden. Dies überlässt dem Benutzer allerdings die Aufgabe gleichzeitige `MPI_Put`-Aufrufe so zu koordinieren, dass keine Fehler auftreten. Es führt zu unvorhersehbaren Daten, wenn die gleiche Speicherstelle in einem Fenster durch zwei gleichzeitige Put-Anweisungen verändert wird. Sobald eine Speicherstelle in einem Fenster von einer RMA-Operation verändert wurde, darf auf diese Stelle keine andere RMA-

Operation zugreifen, bevor der vorherige Transfer nicht vollständig abgeschlossen ist. Dies geschieht erst nach der zugehörigen Synchronisation. Die einzige Ausnahme ist der gleichzeitige Zugriff auf das selbe Fenster durch `MPI_Accumulate`, wo der gleichzeitige Zugriff innerhalb einer Zugriffsepoche durch die Verknüpfung der Daten explizit gewünscht ist. Auf die Reihenfolge der Verknüpfungen kann der Anwender keinen Einfluss nehmen. Allerdings sind alle in MPI vordefinierten Reduce-Operationen kommutativ, wodurch verschiedene Verknüpfungsreihenfolgen bis auf Rundungsfehler zum gleichen Ergebnis kommen sollten.

Ein Prozess darf während des RMA-Zugriffs eines entfernten Prozesses auf ein von ihm freigegebenes Fenster, nicht zeitgleich lokal Fensterpuffer zugreifen. Allerdings darf ein Prozess mit RMA-Operationen auf ein eigenes Fenster zugreifen. Dabei gelten, die für diesen Zeitpunkt gültigen Zugriffsbeschränkungen. Im Kapitel 3.5 wird nach der Erklärung der verschiedenen Synchronisationsmöglichkeiten noch tiefer auf die korrekten Zugriffsmöglichkeiten auf ein Fenster eingegangen.

Zugriffe über die drei RMA-Operationen unterliegen noch weiteren Beschränkungen. Analog zu dem Datentransfer der Punkt-zu-Punkt Kommunikation darf eine `MPI_Put`- bzw. `MPI_Accumulate`-Anweisung nicht über Fenstergrenzen hinweg Daten schreiben, und eine `MPI_Get`-Anweisung darf nicht über Fenstergrenzen hinweg Daten lesen.

3.4 Synchronisation

Um die Zugriffe auf die Fenster zu koordinieren, sind verschiedene Synchronisationsmethoden möglich. Die Synchronisation der einseitigen Kommunikation in MPI kann in zwei Ansätze aufgeteilt werden. Bei der *Synchronisation mit aktivem Ziel* (active target communication) werden die Zielprozesse explizit in die Synchronisation mit einbezogen. D.h. der Anwender muss den Zielprozess auf bevorstehende RMA-Operationen vorbereiten. Dabei reichen die Modelle von einer sehr starken Synchronisation, die den logischen Ablauf der Zugriffe eng mit der tatsächlichen Abfolge der aufgerufenen Funktionen verbindet, bis zu einer lockereren Verbindung zwischen den kommunizierenden Prozessen, die Verschiebungen in dem tatsächlichen Ablauf der Funktionen erlaubt. Bei der *Synchronisation mit passivem Ziel* (passive target synchronisation) ist der Zielprozess nur implizit an der Synchronisation beteiligt und muss durch den Anwender deshalb nicht auf eine bevorstehende RMA-Operation vorbereitet werden.

Bei dem Begriff der Synchronisation ist es wichtig mehrere Aspekte der Synchronisation zu sehen. Bei MPI meint die Synchronisation zum einen den Abgleich der Daten und den korrekten Abschluss einer RMA-Operation, sowie die Synchronisation des Zugriffs verschiedener Prozesse (gegenseitiger Ausschluss). Bei der bereits erwähnten Bibliothek zur einseitigen Kommunikation SHMEM ist eine RMA-Operation abgeschlossen, sobald der Kontrollfluss zum Anwenderprogramm zurückkehrt. Es sind somit nur Funktionen für den gegenseitigen Ausschluss bei SHMEM nötig.

Bei der einseitigen Kommunikation in MPI blockieren die RMA-Operationen nicht bis zum Abschluss des Datentransfers, sondern kehren frühzeitig zum Anwenderprogramm zurück. Der Anwender kann folglich nicht davon ausgehen, dass auf die übertragenen Daten bereits lokal zugegriffen werden kann. Erst die Synchronisation, die eine RMA-Operation umschließt, garantiert abgeglichene Zustände der Daten.

Man kann sich zum besseren Verständnis als logisches Modell eine zusätzliche öffentliche Kopie eines Fensters im Speicher eines Prozesses vorstellen. Der Standard beschreibt dies damit, dass nur eine Instanz jeder Variablen im Prozessspeicher existiert, allerdings eine öffentliche Kopie für jedes Fenster, das diese Variable enthält, zusätzlich geführt wird. Die drei RMA Operationen arbeiten grundsätzlich nur auf diesen Kopien. Lokale Zugriffe und

Datentransfers über die Punkt-zu-Punkt Kommunikation verändern die lokale Instanz der Variablen. Die Veränderung der Kopie kann die lokale Instanz beeinflussen, sowie auch umgekehrt eine Veränderung im lokalen Puffer den Wert einer Variablen in der öffentlichen Kopie beeinflusst. Der Zeitpunkt, wann dies geschieht, ist allerdings nicht vorhersehbar. Er ist nur garantiert abgeschlossen, nachdem eine entsprechende Synchronisationsfunktion für dieses Fenster zurückgekehrt ist.

Grundsätzlich gilt auch, dass innerhalb eines Programms zwischen den einzelnen Synchronisationsmechanismen gewechselt werden kann. Einzelne Zugriffsepochen müssen allerdings abgeschlossen sein, bevor eine neue Zugriffsepoche, mit eventuell anderen Synchronisationsmechanismen, begonnen wird.

Bei der Eröffnung einer Freigabe- oder Zugriffsepoche können MPI noch Parameter in einer `assert`-Variable übergeben werden, welche die Synchronisation beeinflussen können. Diese werden in Kapitel 3.6 weiter beschrieben und in der folgenden Erläuterung der Synchronisationsmechanismen sowie den Beispielen mit 0 angenommen.

3.4.1 Synchronisation mit aktivem Ziel

Bei der Synchronisation mit aktivem Ziel gibt ein Zielprozess explizit sein vorher definiertes Fenster für den entfernten Zugriff frei bzw. schließt es, so dass lokale Zugriffe stattfinden können. In diesem Zusammenhang bezeichnet man den Zeitraum, in dem ein Prozess sein Fenster für RMA-Zugriff freigibt, als *Freigabeepoche* (exposure epoch) und den Zeitraum, in dem ein Prozess auf ein Fenster zugreift, als RMA *Zugriffsepoche* (access epoch). Zwischen Freigabe- und Zugriffsepochen besteht eine 1 : 1 Beziehung, d.h. eine Zugriffsepoche gehört nur zu genau einer Freigabeepoche und eine Freigabeepoche ist nur für eine bestimmte Zugriffsepoche offen. Ein Prozess, der seine Zugriffsepoche auf ein Fenster bereits abgeschlossen hat, kann also nicht noch einmal während der selben Freigabeepoche auf das Fenster zugreifen. Er muss warten bis der Zielprozess das Fenster erneut freigibt. Da das MPI Subsystem nicht effizient feststellen kann, ob Fensterpuffer lokal verändert wurden, bezeichnen alle Zeiträume zwischen zwei Synchronisationen, in denen kein RMA Zugriff erfolgt, eine lokale Zugriffsepoche.

Starke Synchronisation mit Fence

Mit `MPI_Win_fence` hat der Anwender die Möglichkeit, die Prozesse und ihre Zugriffe auf Fenster sehr stark zu synchronisieren. Die Bedeutung des Wortes Fence verdeutlicht die Art der Synchronisation. RMA-Operationen werden durch Aufrufe von `MPI_Win_fence` eingezäunt und erreichen so zusätzlich eine zeitliche Synchronisation zwischen den Prozessen. Dies wird durch eine implizite Barriere innerhalb des Aufrufs erreicht, die den Kontrollfluss auf keinem beteiligten Prozess an das Anwendungsprogramm zurückgibt, bevor nicht jeder der Prozesse den Aufruf von `MPI_Win_fence` begonnen hat. Ein Schema einer solchen Synchronisation ist in Abbildung 3.1 dargestellt. Da Fence als einzige Synchronisationsfunktion kollektiv über die ganze Gruppe, die einem Fenster zugeordnet ist, abläuft, bedeutet dies natürlich bei großen Gruppen viele Möglichkeiten für Wartezeiten der einzelnen Prozesse. In Kapitel 6 wird darauf noch speziell hingewiesen, da sich dafür eigene Leistungsmerkmale definieren lassen. Bei einem Aufruf wird nur das Fenster angegeben, auf das zugegriffen werden soll, und nicht ein spezieller Prozess. Der eigentliche Zielprozess wird erst mit der folgenden RMA-Operation angegeben. Der Aufruf bietet somit dem Anwender eine große Flexibilität bei sehr einfacher Handhabung.

Aufgrund der übergreifenden Synchronisation ist es sinnvoll Fence nur dort einzusetzen, wo diese starke zeitliche Synchronisation erforderlich ist und nicht anderweitig, wie zum Beispiel durch effiziente Lastverteilung, hergestellt werden kann. Ebenfalls sinnvoll ist die Benutzung

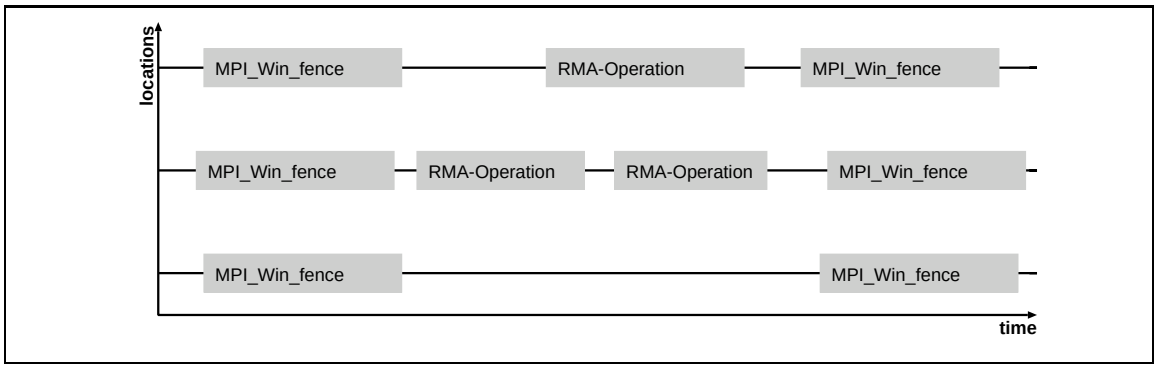


Abbildung 3.1: Synchronisation mit Fence

eines Fence bei vielen, sich oft ändernden Kommunikationspartnern, da in diesem Fall die später erklärte schwache Synchronisation durch den Aufruf von vorbereitenden Funktionen einigen Overhead verursachen kann, und die ebenfalls später beschriebene Synchronisation mit passivem Ziel die Prozesse eventuell zu stark auseinander driften lässt.

Die Synchronisation mit Fence ermöglicht ein einfaches Zugriffsmuster innerhalb eines Programms: globale Datenaustauschsabschnitte lösen sich mit globalen Berechnungsabschnitten ab. Global meint in diesem Zusammenhang den Kommunikator auf dem ein Fenster definiert ist. Dabei werden Speicherfenster und lokale Variablen bei jedem Wechsel zwischen Datenaustausch und Berechnung durch ein Fence synchronisiert. Beispiel 3.1 zeigt eine mögliche Abfolge der Synchronisation.

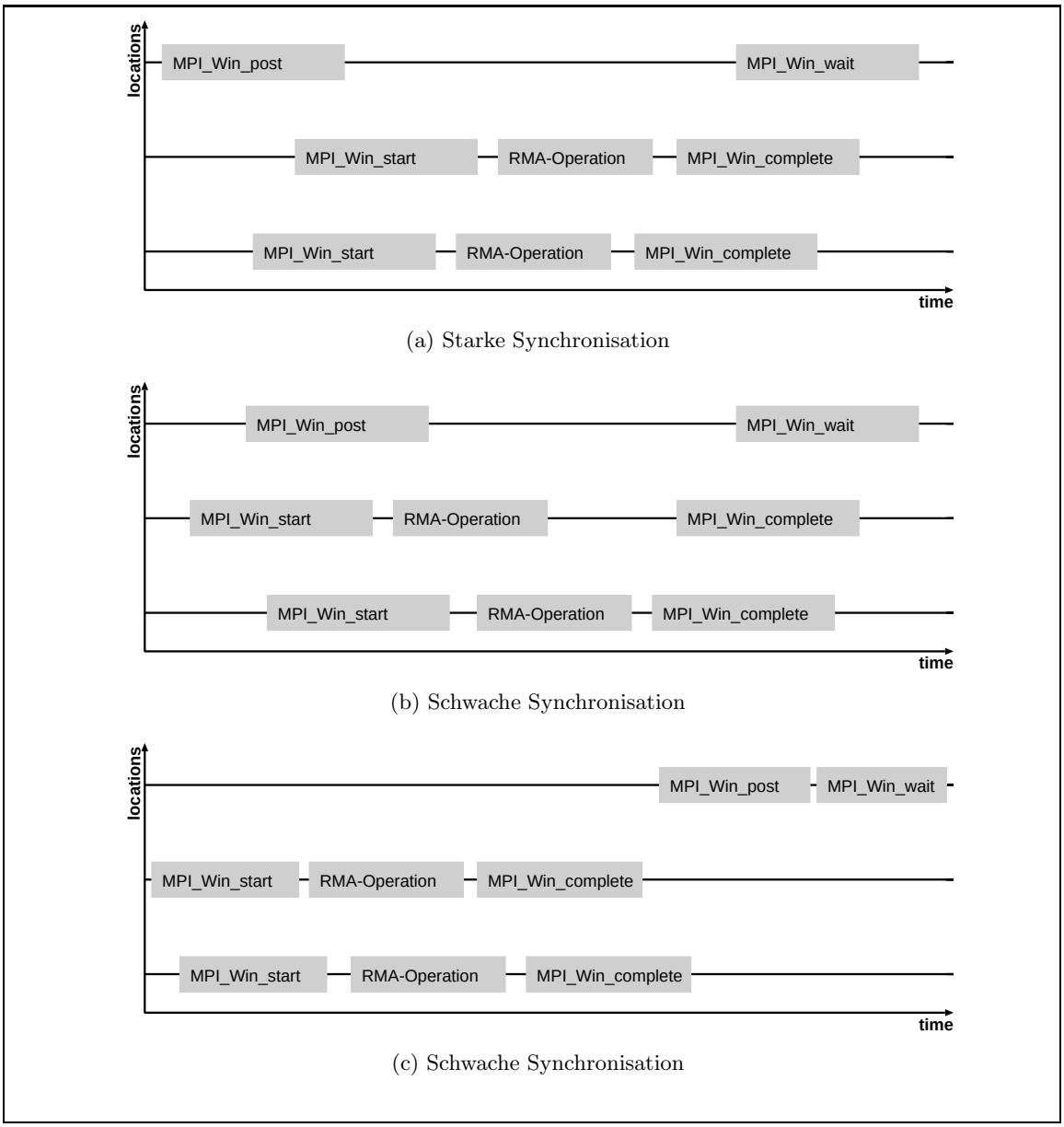
```
/* Berechnung */

MPI_Win_fence(0, win);

/* Datenaustausch */
MPI_Put(..., target_rank, ..., win);

MPI_Win_fence(0, win);

/* Berechnung */
```

Beispiel 3.1: Synchronisation mit `MPI_Win_fence`

Zusammenfassend schließt ein Fence eine offene Epoche für ein Fenster ab und eröffnet potenziell eine neue Epoche. Implizit können dadurch keine zwei Freigabeepochen für das gleiche Fenster existieren. Zugriffs- bzw. Freigabeepochen verschiedener Fenster können allerdings überlappend sein.

Allgemeine Synchronisation mit aktivem Ziel

Bei der allgemeinen Synchronisation mit aktivem Ziel ist es implementationsabhängig, ob eine *starke Synchronisation* (strong synchronization) oder eine *schwache Synchronisation* (weak synchronization) zwischen den teilnehmenden Prozessen vorherrscht. Bei der schwachen Synchronisation erlaubt MPI den Prozessen eine zeitliche Abweichung von dem logischen Ablauf der Ereignisse. Genauere Beschreibungen dazu werden später im Kapitel noch einmal angeführt. Die Synchronisation besteht aus fünf verschiedenen Funktionen, um Zugriffs- und Freigabeepochen mit großer Flexibilität angeben zu können und die Synchronisation zwischen den Prozessen auf ein Minimum zu beschränken. Der Benutzer erhält



(a) Starke Synchronisation

(b) Schwache Synchronisation

(c) Schwache Synchronisation

Abbildung 3.2: Schemata der allgemeinen Synchronisation mit aktivem Ziel

damit ein etwas komplexeres Synchronisationsmodell, welches mehr Kontrolle und Möglichkeiten bietet auf die vorgenommene Synchronisation Einfluss zu nehmen.

Eine Freigabeepoche wird mit `MPI_Win_post` gestartet und mit `MPI_Win_wait` oder `MPI_Win_test` beendet. `MPI_Win_wait` blockiert dabei solange bis alle zugreifenden Prozesse ihre Zugriffsepoche beendet haben. `MPI_Win_test` blockiert nicht und gibt in einem Flag zurück, ob ein Aufruf von `MPI_Win_wait` zu diesem Zeitpunkt abgeschlossen wäre oder nicht. Solange noch nicht alle Zugriffe abgeschlossen sind, enthält dieses Flag den Wert 0 und der Aufruf hat keinen weiteren Einfluss auf das Programm. Sobald das Flag einen Wert ungleich 0 zurückliefert, ist die Freigabeepoche auch direkt abgeschlossen, und weder ein `MPI_Win_wait` noch ein weiteres `MPI_Win_test` dürfen folgen, ohne dass eine neue Freigabeepoche gestartet wird. Auf den zugreifenden Prozessen muss eine entsprechende Zugriffsepoche durch `MPI_Win_start` eröffnet werden und mit `MPI_Win_complete` abgeschlossen werden. Diese explizite Angabe der Aufrufe lässt eine genauere Aufschlüsselung der Zugriffs- und Freigabeepochen schon zur Laufzeit zu. Wie später ersichtlich wird, ist dies auch nötig, da sich bei der schwachen Synchronisation zeitliche Verschiebungen bei den

Epochen ergeben können.

Ein Vorteil der allgemeinen Synchronisation, im Gegensatz zur Synchronisation mit Fence, bildet die Tatsache, dass sich hierbei nur die Prozesse paarweise synchronisieren, die auch tatsächlich an einer RMA-Operation beteiligt sind. So wird der Synchronisationsoverhead minimiert. Der Synchronisation werden Gruppen übergeben, welche alle teilnehmenden Prozesse beinhalten müssen. D.h., der Zielprozess übergibt `MPI_Win_post` die Gruppe mit allen Ursprungsprozessen für die nächste Freigabeepoche. Jeder Ursprungsprozess übergibt `MPI_Win_start` alle Zielprozesse der damit gestarteten Zugriffsepoche. Jeder Zielprozess weiß von welchem Prozess Zugriffe zu erwarten sind. Die einzelnen Gruppen, die den Funktionen übergeben werden, müssen nicht identisch sein. Es werden aus der Gruppe aller Prozesse, die dieses Fenster mit definiert haben, kleine Untergruppen, welche genau auf den kommenden Datentransfer ausgelegt sind, gebildet. Sich häufig ändernde Kommunikationspartner würden dabei zusätzlichen Overhead erzeugen, da häufig neue Gruppen erzeugt werden müssten. Die Aufrufe zur Gruppenerzeugung bringen zusätzliche unproduktive Zeit in den Ablauf des Programms, weshalb es empfehlenswert ist, einmal die Kommunikationsgruppen zu definieren und diese möglichst nicht mehr zu verändern.

Beispiel 3.2 veranschaulicht die Arbeitsweise der allgemeinen Synchronisation mit aktivem Ziel. Dabei gibt der erste Teil einen Eindruck über den Aufwand der Vorbereitungen zu dieser Synchronisation, da die Prozesse erst in verschiedene Gruppen aufgeteilt werden müssen. Die Gruppe `target_group` besteht dabei aus den Prozessen in `target_ranks`. `origin_group` besteht aus allen anderen Prozessen.

```
/* Teile alle Prozesse in Gruppen von 2 Prozessen auf */
MPI_Comm_split(MPI_COMM_WORLD, (int)(rank / 2), rank, &comm);
MPI_Comm_group(comm, &comm_group);
MPI_Comm_rank(comm, &group_rank);
MPI_Group_incl(comm_group, 1, &target_rank, &target_group);
MPI_Group_difference(comm_group, target_group, &origin_group);

MPI_Comm_group(MPI_COMM_WORLD, &world_group);
MPI_Group_translate_ranks( target_group, 1, &target_rank,
              world_group, &global_target_rank );

if (group_rank == target_rank)
{
  /* am Ziel: Freigabeepoche abwickeln */
  MPI_Win_post(origin_group, 0, win);
  MPI_Win_wait(win);
}
else
{
  /* am Ursprung: Zugriffsepoche abwickeln */
  MPI_Win_start(target_group, 0, win);
  MPI_Put(..., global_target_rank, ..., win);
  MPI_Win_complete(win);
}
```

Beispiel 3.2: Allgemeine Synchronisation mit aktivem Ziel

Der Name der schwachen Synchronisation kommt von der leichten Verschiebung im tatsächlichen Ablauf, welche mit MPI erlaubt sind. Funktionen müssen nicht zwangsläufig in

ihrer logischen Reihenfolge, die in Abbildung 3.2(a) dargestellt ist, in der Realität auftreten. MPI gibt bei der Synchronisation von RMA-Operationen aus Effizienzgründen nur vor, dass nicht mehr als eine Zugriffsepoche bei einem Ursprungsprozess abgeschlossen sein darf, bevor die zugehörige Freigabeepoche auf dem Zielprozess begonnen wird. Dies gibt einer Implementation einige Freiheit, wann auf dem Ursprungsprozess eine Blockierung erfolgen kann bzw. muss. Eines der folgenden Modelle ist möglich:

a) `MPI_Win_start` blockiert auf dem Ursprungsprozess solange, bis der Zielprozess den Aufruf von `MPI_Win_post` getätigt hat.
b) `MPI_Win_start` blockiert nicht, `MPI_Put` blockiert auf dem Ursprungsprozess bis der Zielprozess den Aufruf von `MPI_Win_post` getätigt hat.
c) `MPI_Win_start` und `MPI_Put` blockieren nicht, aber `MPI_Win_complete` blockiert, bis der Zielprozess den Aufruf von `MPI_Win_post` getätigt hat.
d) Keine der drei Funktionen blockiert auf dem Ursprungsprozess, die Daten werden gepuffert, bis der Zielprozess den Aufruf von `MPI_Win_post` getätigt hat.

Somit kann die Implementation bei der schwachen Synchronisation einen Verlauf der Funktionsaufrufe zeigen, wie in Abbildung 3.2(b) und Abbildung 3.2(c) angedeutet. Durch solche Verschiebungen können Situationen innerhalb der Kommunikation zwischen den Prozessen auftreten, in denen ein Prozess schon eine Zugriffsepoche startet, noch bevor der Zielprozess die vorherige Freigabeepoche abgeschlossen hat.

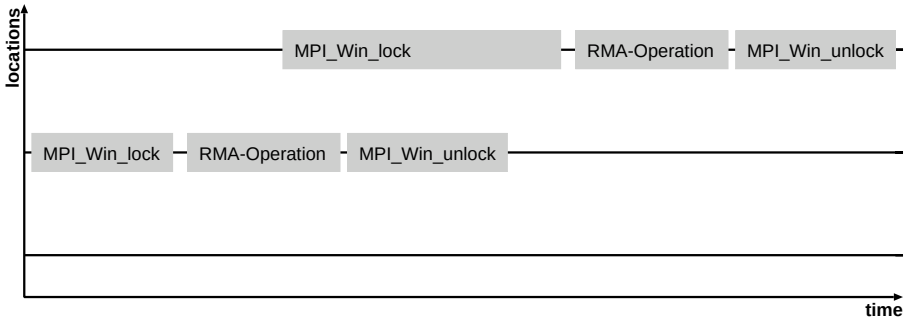
Diese Freiheit im Standard lässt für die Hersteller viele Möglichkeiten einer Implementierung offen, um eine effiziente Lösung zu finden. Allerdings erschwert es die Definition von Leistungsmerkmalen und eine spätere Analyse des Programms.

3.4.2 Synchronisation mit passivem Ziel

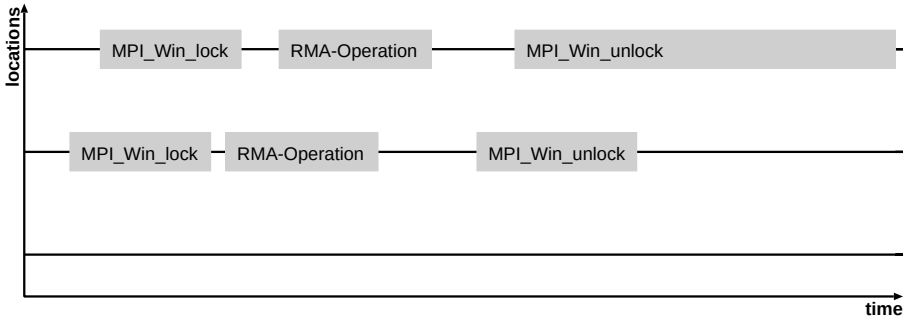
Wenn der Zielprozess einer RMA-Operation sich nicht an der Synchronisation beteiligt, sondern die Ursprungsprozesse selbst die Freigabeepochen bestimmen, spricht man von *Kommunikation mit passivem Ziel* (passive target communication). Zwar stellt der bereitstellende Prozess Sperren zur Verfügung, mit denen die Synchronisation bewerkstelligt wird, aber er ruft keine expliziten Synchronisationsfunktionen auf. Dies ist die dritte und letzte Möglichkeit innerhalb MPI-2 eine einseitige Kommunikation zu synchronisieren. Hierbei melden die Ursprungsprozesse mittels `MPI_Win_lock` an, dass sie Zugriff auf ein Fenster des Zielprozesses haben wollen. Nach der RMA-Operation wird die Sperre mit `MPI_Win_unlock` wieder entfernt. Auch hier gibt der MPI Standard viel Raum für eine effiziente Implementation bezüglich des blockierenden Verhaltens der Synchronisationsfunktionen. So kann `MPI_Win_lock` blockieren bis der Zugriff auf das Fenster wirklich gewährt werden kann. Ebenso ist es möglich, dass in einer konkreten Implementation weder `MPI_Win_lock` noch ein folgender `MPI_Put`-Aufruf blockiert, und erst `MPI_Win_unlock` solange blockiert, bis die gesamte Operation abgeschlossen ist. Daraus resultieren wiederum Schwierigkeiten für ein klares Modell des Datenaustauschs, wie später noch ersichtlich wird. Freigabe- und Zugriffsepoche sind nun allerdings nicht komplett kongruent, wie es zu vermuten wäre. Der MPI Standard gibt an, dass `MPI_Win_lock` eine Zugriffsepoche startet und `MPI_Win_unlock` diese beendet. Gleichzeitig beendet `MPI_Win_unlock` auch die Freigabeepoche für den Ursprungsprozess. In der Situation, in der `MPI_Win_lock` und `MPI_Put` nicht blockieren und erst `MPI_Win_unlock` blockiert, bis die RMA-Operation auf beiden Seiten abgeschlossen ist, kann ein genauer Zeitpunkt für den Anfang der Freigabeepoche nicht direkt gegeben werden.

Wie der Name der Synchronisationsfunktionen es auch andeutet, arbeitet die passive Synchronisation mit Sperren. Allerdings ist die intuitive Annahme, dass nach dem Aufruf von `MPI_Win_lock` das Fenster für den Prozess gesperrt ist, nicht garantiert. `MPI_Win_lock`

muss nicht blockieren bis die Sperre erfolgreich gesetzt ist. Daher ist der Aufruf mehr als Anforderung einer Sperre zu verstehen, genau wie eine nicht blockierende RMA-Operation auch nur eine Anforderung zum Datentransfer darstellt. `MPI_Win_unlock` verarbeitet in diesem Fall sowohl die Zuteilung der Sperre, den Datentransfer sowie die anschließende Freigabe der Sperre. Mögliche Synchronisationsabläufe sind in Abbildung 3.3 dargestellt. Abbildung 3.3(a) zeigt einen Ablauf, wie er bei einer exklusiv gesetzten Sperre durch eine blockierende Lock-Funktion erzeugt werden könnte. Abbildung 3.3(b) zeigt einen Ablauf der bei einem nicht-blockierenden Aufruf von `MPI_Win_lock` auftreten kann.



(a) Synchronisation mit passivem Ziel



(b) Synchronisation mit passivem Ziel (nicht-blockierendes `MPI_Win_lock`)

Abbildung 3.3: Schemata der Synchronisation mit passivem Ziel

Bei der Synchronisation mit Sperren kann eine *gemeinsame* (shared) oder eine *exklusive* (exclusive) Sperre gesetzt werden. Eine exklusive Sperre gewährt einem Prozess Schreib- und Lesezugriff auf das gegebene Fenster. Keine anderen Sperren können gleichzeitig anderen Prozessen für dieses Fenster zugeteilt werden. Eine gemeinsame Sperre gewährt einem Prozess den Lesezugriff auf ein Fenster. Andere Prozesse können ebenfalls eine gemeinsame Sperre setzen, um von dem Fenster Daten zu lesen. Dadurch ist es möglich, quasigleichzeitige Leseoperationen auf einem Fenster zu haben. Während eine gemeinsame Sperre gesetzt ist, darf keine exklusive Sperre gesetzt werden. Die Art der Sperre wird dem Aufruf von `MPI_Win_lock` übergeben. Dabei sind `MPI_LOCK_EXCLUSIVE` für eine exklusive und `MPI_LOCK_SHARED` für eine gemeinsame Sperre möglich.

`MPI_Put` benötigt immer eine exklusive Sperre des Fensters, da es keinem anderen Prozess gestattet werden kann, während der Datenübertragung auf das Fenster lesend oder schreibend zuzugreifen. `MPI_Accumulate` kann sowohl mit gemeinsamer als auch mit exklusiver Sperre benutzt werden, obwohl die für RMA zusätzlich definierte Reduce-Operation `MPI_REPLACE` mit gemeinsamen Sperren nicht sinnvoll ist, da sich aufgrund der beliebigen Reihenfolge bei der Ausführung der Reduce-Operation am Ende ein nicht definiertes Ergebnis ergibt. `MPI_Get` benötigt hingegen nur eine gemeinsame Sperre. Es empfiehlt sich die Art der Sperre an die gewünschte Operation anzupassen. Z.B. führen mehrere durch exklusive Sperren synchronisierte Lesezugriffe auf ein Fenster zu einer seriellen Ausführung

der Operationen und damit zu unnötigen Wartezeiten.

Diese Art der Synchronisation hat die geringfügigste gegenseitige Kopplung und Abstimmung der Prozesse innerhalb der einseitigen Kommunikation mit MPI. Dies bedeutet oft höhere Effizienz und geringere Wartezeiten der Prozesse, allerdings auch gerade bei exklusivem Zugriff ein zeitliches Auseinanderdriften der einzelnen Prozesse. Um dies zu verhindern muss der Anwender an wichtigen Stellen selbst Synchronisationspunkte, z.B. durch eine Barriere, im Programm setzen. Das Beispiel 3.3 zeigt eine einfache Situation auf, bei der die Prozesse vor der Ausgabe neu synchronisiert werden müssen.

Jeder Prozess speichert mit `MPI_Put` einen `double`-Wert im entsprechenden Fenster von Prozess 0 mit einem *Versatz* (displacement) in Höhe seines jeweiligen Ranges ab. Ohne die explizite Barriere würde Prozess 0, nachdem `MPI_Win_unlock` den Kontrollfluss zurückgibt, direkt die Ausgabe starten, unabhängig von der Anzahl der Prozesse, die noch auf die Freigabe des Fensters warten. Auf diese Weise würden nur die Änderungen im Fenster, die vor der Übertragung von Prozess 0 abgeschlossen waren, angezeigt. Erst die Barriere sorgt dafür, dass alle Prozesse aus `MPI_Win_unlock` zum Anwenderprogramm zurückgekehrt sind, bevor die Ausgabe von Prozess 0 beginnt.

Wird einseitige Kommunikation mit einer der ersten beiden Synchronisationskonzepte benutzt, ist dies noch nicht vollständig einseitig. Es erfordert die aktive Teilnahme an der Synchronisation. Die passive Synchronisation wird jedoch ihrem Ziel, der vollständig einseitigen Kommunikation, gerecht. Zielprozesse müssen nicht mehr vom Anwender in die Zugriffsabfolge eingebunden werden.

```
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, win);
MPI_Put(&var, 1, MPI_DOUBLE, 0, rank, 1, MPI_DOUBLE, win);
MPI_Win_unlock(win);

MPI_Barrier(MPI_COMM_WORLD);

if ( my_rank == 0 )
  for ( i = 0; i < comm_size; i++ )
    printf("buffer[%i]: %.6lf\n", i, buffer[i] );
```

Beispiel 3.3: Ausschnitt einer Synchronisation mit Sperren

3.5 Korrekter RMA-Zugriff

Durch die einseitige Natur der Aufrufe stellt sich ein wichtige Frage: Wann kann ein RMA-Aufruf als abgeschlossen gelten? Im Kapitel 4 benutzt das erweiterte Ereignismodell diesen Zeitpunkt des Abschlusses einer RMA-Operation als Zeitpunkt zur Modellierung des Datentransferendes. Der Standard gibt hierzu verschiedene Grundregeln an, die im folgenden Abschnitt und in Abbildung 3.4 näher ausgeführt werden.

`MPI_Put` kann den Kontrollfluss schon an das Benutzerprogramm zurückgeben, sobald der Datentransfer erfolgreich vorbereitet wurde. Die eigentliche Übertragung wird im Hintergrund abgeschlossen. Die Rückkehr des Kontrollflusses zum Anwenderprogramm beinhaltet also keinerlei Aussage über die Abgeschlossenheit der Operation. Um ein konsistentes Verhalten und definierte Zustände zu ermöglichen, werden im MPI-Standard [12] folgende Regeln aufgestellt:

R1 Eine RMA-Operation gilt dann am Ursprung als abgeschlossen, wenn der darauf folgende Aufruf von `MPI_Win_fence`, `MPI_Win_complete` oder `MPI_Win_unlock`, der diese Operation synchronisiert, vom Ursprungsprozess verlassen wird.

R2 Wenn eine RMA-Operation am Ursprung durch `MPI_Win_fence` abgeschlossen ist, so wird sie durch den zugehörigen Aufruf von `MPI_Win_fence` auf dem Zielprozess für den Zielprozess abgeschlossen.

R3 Wenn eine RMA-Operation am Ursprung durch `MPI_Win_complete` abgeschlossen ist, so wird sie am Ziel durch den zugehörigen Aufruf von `MPI_Win_wait` auf dem Zielprozess für den Zielprozess abgeschlossen.

R4 Wenn eine RMA-Operation am Ursprung durch `MPI_Win_unlock` abgeschlossen ist, so gilt die Operation am Ziel durch den selben Aufruf von `MPI_Win_unlock` als abgeschlossen.

R5 Eine Veränderung einer Speicherstelle im lokalen Puffer eines Prozesses wird spätestens dann in der öffentlichen Kopie des Fensters sichtbar, wenn ein darauf folgender Aufruf von `MPI_Win_fence`, `MPI_Win_post` oder `MPI_Win_unlock` vom Besitzer des Fensters ausgeführt wurde.

R6 Eine Veränderung der öffentlichen Kopie durch `MPI_Put` oder `MPI_Accumulate` wird im lokalen Speicher des Fensters spätestens dann sichtbar, wenn der darauf folgende Aufruf von `MPI_Win_fence`, `MPI_Win_wait` oder `MPI_Win_unlock` vom Besitzer des Fensters ausgeführt wurde.

Diese Regeln führen zu dem in Abbildung 3.4 schematisch dargestellten Zugriffsschema. In den Regeln R3 und R6 wird ein Aufruf von `MPI_Win_test` mit angezeigtem Abschluss der Freigabeepoche wie ein Aufruf von `MPI_Win_wait` behandelt. In der Abbildung wird ersichtlich, dass nach einem verändernden Zugriff auf Daten, ein Abgleich zwischen der lokalen und der öffentlichen Kopie eines Fensters nur durch eine Synchronisation erreicht werden kann. Die Pfeile geben dabei die Richtung des Datentransfers an. Eine Pfeilspitze impliziert eine Veränderung der Daten. Eine durchgezogene Linie stellt eine garantierte Veränderung dar. Eine gestrichelte Verbindung zeigt an, dass die Operation die entsprechenden Daten beeinflussen kann, aber nicht muss.

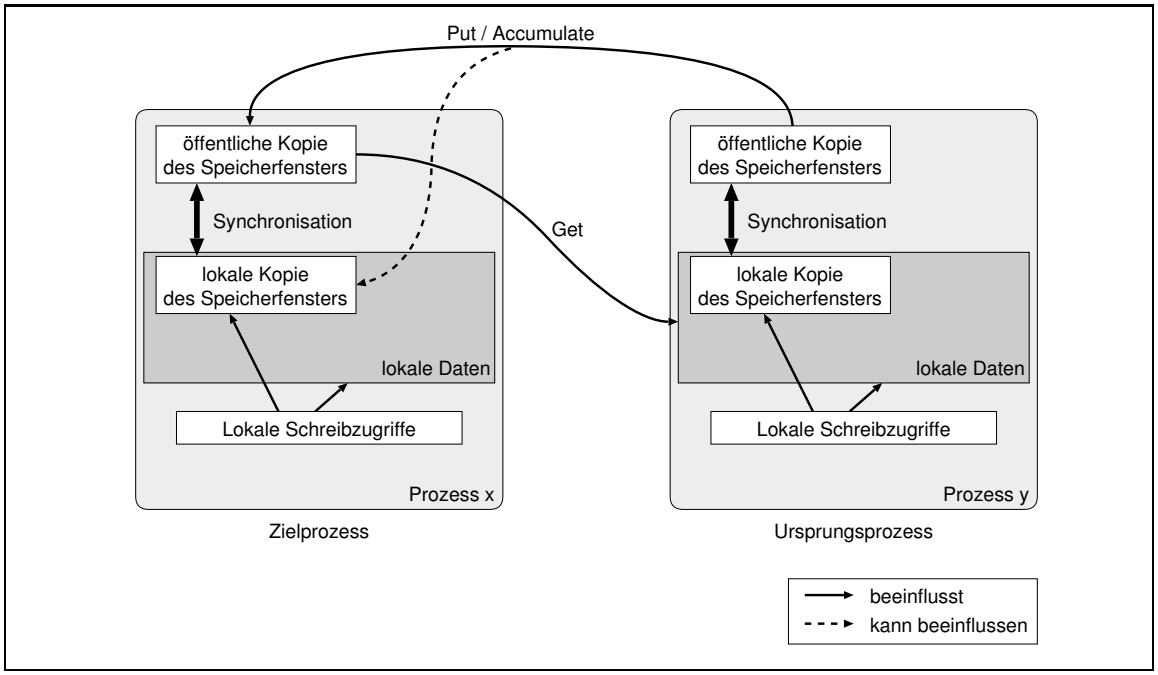


Abbildung 3.4: Logisches Schema für korrekte RMA-Zugriffe bei MPI

Diese vom MPI Forum angebrachten Regeln sind teilweise so offen für Hersteller von Implementationen, dass eine RMA-Operation auf unterschiedlichen Plattformen sich sehr verschieden bezüglich der Abgeschlossenheit verhalten kann. Dies kann sowohl die Suche nach Gründen für geringe Leistung als auch die Behebung dieser durch den Anwender er-

schweren. R4 gibt an, dass die Operation auch für den Zielprozess durch den Aufruf von `MPI_Win_unlock` als abgeschlossen gilt, allerdings erlaubt der Standard ebenfalls eine Verzögerung des Abgleichs zwischen öffentlicher Kopie des Fensters und lokalem Prozessspeicher bis zur nächsten Synchronisationsfunktion auf dieses Fenster durch den Besitzer des Fensters. Das Programmierungsschema bei der Synchronisation mit passivem Ziel sieht es also vor, dass, egal ob es sich um einen lokalen oder entfernten Zugriff auf Daten innerhalb eines Speicherfensters handelt, alle Prozesse den Zugriff über die Funktionen `MPI_Win_lock` und `MPI_Win_unlock` synchronisieren. Da der Prozess, der das Speicherfenster bereitstellt nicht an der explizit an der Synchronisation anderer Zugriffe als seiner eigenen beteiligt ist, ist dies die einzige Möglichkeit Datenkonsistenz zu gewährleisten.

3.6 Assertions - Einflussnahme auf die Synchronisation

Bei jeder RMA Synchronisationsfunktion, die eine Zugriffsepoche oder eine Freigabeepoche startet, kann der Funktion ein Wert übergeben werden, der dem System erlaubt, gewisse Annahmen bei der Verarbeitung der Synchronisation vorauszusetzen, um dadurch bestimmte Zugriffe oder Abgleiche zu sparen. So kann mit `MPI_MODE_NOCHECK` vom Anwender explizit bei `MPI_Win_start` angegeben werden, dass das zugehörige `MPI_Win_post` auf dem Zielprozess bereits ausgeführt wurde, und somit Situationen wie in Abbildungen 3.2(b) und 3.2(c) vom Anwender ausgeschlossen werden. Von der Anschauung ist dies ähnlich zu dem Aufruf von `MPI_Rsend`, welches ein bereits gestartetes `MPI_Recv` auf dem Zielprozess voraussetzt. Umgekehrt gibt diese Option bei `MPI_Win_post` an, dass von keinem zugreifenden Prozess bereits ein `MPI_Win_start` getätigt wurde. Bei einem Aufruf von `MPI_Win_lock` gibt diese Option an, dass kein anderer Prozess momentan eine Sperre auf das angegebene Fenster besitzt. Insgesamt kann diese Option Sicherheitsabfragen vermeiden, die unnötige Wartezeiten verursachen würden.

Mit `MPI_MODE_NOSTORE` kann der Anwender bei `MPI_Win_post` und `MPI_Win_fence` explizit angeben, dass das Fenster seit der letzten Fensteroperation nicht durch eine lokale STORE-Anweisung verändert wurde. Dazu gehören auch `MPI_Get`-Operationen und Receive-Operationen, die einen Empfangspuffer beinhalten, der in dem Fenster freigegeben ist. Dies kann unnötige Cache-Abgleiche zu Beginn einer Freigabeepoche sparen.

Durch die Option `MPI_MODE_NOPUT` kann bei `MPI_Win_post` und `MPI_Win_fence` angegeben werden, dass das Fenster nach dem Aufruf bis zum Ende der Freigabeepoche nicht durch eine RMA-Operation verändert wurde. Dies kann wiederum Cache-Abgleiche am Ende einer Freigabeepoche ersparen.

`MPI_MODE_NOPRECEED` und `MPI_MODE_NOSUCCEED` nehmen sich schließlich dem am Anfang erwähnten Problem an, bei einem Aufruf von `MPI_Win_fence` festzustellen ob gerade eine RMA-Epoche anfängt, endet oder beides. `MPI_MODE_NOPRECEED` gibt dabei an, dass der Aufruf keine RMA-Operationen abschließen muss, und `MPI_MODE_NOSUCCEED` gibt an, dass der Aufruf keine RMA-Operationen vorbereiten muss.

Die Optionen und ihre Bedeutung sind im Standard festgehalten, allerdings ist es den Herstellern einer MPI Implementation explizit freigestellt, ob und welche der Optionen implementiert werden. Eine 0 als Assert-Parameter muss jede Implementation als pessimistischsten Fall annehmen und somit alle nötigen Synchronisationen und Abgleiche durchführen. Gleichzeitig ist eine Implementation daran gebunden, eine nicht implementierte Option bei dem Aufruf zu ignorieren, so dass Anwender diese Optionen nutzen können, ohne Kohärenzprobleme zu bekommen. Durch die Angabe dieser Optionen können Wartezeiten durch unnötige Datenabgleiche vermieden werden, allerdings sind bei falschen Angaben Dateninkonsistenzen zu erwarten.

Kapitel 4

Ereignismodelle für die einseitige Kommunikation

4.1 Motivation

Alle Einzelheiten des dynamischen Ablaufs eines Programms aufzuzeichnen, benötigt viel Speicherplatz und Zeit. Daher muss eine Abstraktion geschaffen werden, um von einem komplexen Prozessablauf zu einer möglichst einfachen Beschreibung zu gelangen, ohne essenzielle Eigenschaften des Prozessablaufs zu verlieren. Dazu wird im folgenden ein Ereignismodell definiert, dass alle leistungsrelevanten Ereignisse des Programmablaufs abbilden kann.

Im Bereich der nachrichtenbasierten Prozesskommunikation mit MPI konnte durch KOJAK bisher nur die Kommunikation im Anwenderprogramm modelliert werden, die dem MPI-Standard 1.2 entsprach. Das Kommunikationskonzept der einseitigen Kommunikation in MPI 2.0 ist von den in MPI 1.2 beschriebenen Kommunikationskonzepten so verschieden, dass die dafür zur Verfügung stehenden Mittel zur Modellierung der Abläufe nicht ausreichen. Es muss ein Ereignismodell entwickelt werden, mit dem die Ereignisse der einseitigen Kommunikation zufriedenstellend modelliert werden können, um eine spätere Analyse zu ermöglichen. Das entwickelte Modell muss sich mit seinen Eigenschaften in die bisherige Modellierung der dynamischen Abläufe eines Programms einfügen. Neben der automatischen Analyse werden Ereignisströme für die manuelle Analyse oft visualisiert, um dem Anwender die Abläufe in seinem Programm grafisch zu verdeutlichen. Eine mächtige und etablierte Werkzeugumgebung zur Visualisierung von Ereignisströmen der ereignisbasierten Leistungsanalyse ist VAMPIR. VAMPIR verwendet in seiner Aufzeichnungsbibliothek VAMPIRTRACE für die einseitige Kommunikation ein, dem später beschriebenen Basismodell, sehr ähnliches Modell. Dies ist einer der Gründe für die Unterstützung dieses Modells innerhalb der Aufzeichnung durch EPILOG. So kann ein Ereignisstrom erzeugt werden, der mit der Modellierung der VAMPIR-eigenen Aufzeichnung vergleichbar ist. Eine Ereignisspur im EPILOG-Format lässt dabei aber detailliertere Rückschlüsse über den dynamischen Ablauf des Programms zu. Da Programmläufe großer Projekte unter Umständen eine Laufzeit von mehreren Stunden haben können, ist es an dieser Stelle sinnvoll, wenn beide Arten der Analyse genutzt werden sollen, die Originalaufzeichnung durch KOJAK zu tätigen und dann für die Visualisierung eine Konvertierung vorzunehmen, als zwei unabhängige Programmläufe, einmal durch KOJAK und einmal durch VAMPIR, aufzuzeichnen. Die verschiedenen Programmläufe würden dann auch vielleicht ein ähnliches Verhalten zeigen, aber sich nicht direkt entsprechen.

Das Basismodell für die einseitige Kommunikation beschreibt zunächst eine einfache Sicht

auf die relevanten Vorgänge. Es belässt Ereignisse an dem Zeitpunkt an dem sie aufgezeichnet wurden. Dies ist leicht zu implementieren und kann oft für die spätere Analyse ausreichend sein. In manchen Fällen stimmen der Aufzeichnungszeitpunkt und der tatsächliche Zeitpunkt des Ereignisses, welches man modellieren möchte, jedoch nicht gut genug überein. In diesen Fällen ist das Basismodell zu ungenau und es macht die Entwicklung eines erweiterten Modells notwendig, das die Zeitpunkte der Ereignisse im Ereignisstrom näher mit den realen Zeitpunkten der Ereignisse des Anwenderprogramms verbindet. Das erweiterte Modell berücksichtigt die vom MPI-Standard beschriebene Semantik der Synchronisation, um den einseitigen Datentransfer innerhalb des Ereignisstroms relativ zum spätesten Zeitpunkt seiner Abgeschlossenheit zu modellieren. Dadurch wird erreicht, dass die modellierten Ereignisse näher an dem tatsächlichen Zeitpunkt des Auftretens liegen.

Durch eine Umsetzung der in EPILOG aufgezeichneten Ereignisströme auf ein VAMPIR-eigenes Spurformat können die Vorgänge der einseitigen Kommunikation durch VAMPIR sowohl im Basismodell, als auch im erweiterten Modell dargestellt werden. Implizit vorhandene Informationen innerhalb des Ereignisstroms, sind für den Anwender in der grafischen Darstellung aber unter Umständen nicht zu erschließen. Deshalb kann es notwendig sein, das erweiterte Ereignismodell stärker auf die Bedürfnisse bei der Visualisierung anzupassen. Dazu werden in Kapitel 7 einige Ansätze gegeben.

4.2 Einführung in die Ereignismodelle der Leistungsanalyse

In der ereignisbasierten Leistungsanalyse und insbesondere im Tracing werden die dynamischen Abläufe der zu analysierenden Programme durch eine Folge von Ereignissen ausgedrückt. Um die in Kapitel 2 beschriebenen Nachteile des Tracings zu minimieren, muss darauf geachtet werden, den verfügbaren Speicherplatz möglichst optimal zu nutzen. Dazu müssen Anzahl und Art der verwendeten Ereignistypen so beschaffen sein, dass sie ein Minimum an benötigtem Speicherplatz in der erzeugten Spur mit einer maximalen Flexibilität bei der Definition und Suche nach Leistungsmerkmalen verbinden. Das bedeutet, dass ein erhöhter Speicherplatzverbrauch gerechtfertigt werden kann, wenn dadurch ein erheblicher Nutzen für die Analyse entsteht. Aufgrund der unüberschaubaren Fülle an zur Laufzeit anfallenden Prozessdaten wird nur eine Untermenge dieser Daten mit Hilfe von Ereignissen modelliert. Deshalb erlaubt der Ereignisstrom nur eine eingeschränkte Sicht auf die tatsächlichen Abläufe innerhalb des Programms. Die wichtigsten auf die Leistung Einfluss nehmenden Ereignistypen werden durch ein entsprechendes Ereignismodell beschrieben. Ereignisse in der aufgezeichneten Spur sind Instanzen dieser Ereignistypen, d.h. ein Ereignis innerhalb des Ereignisstroms ist eine Datenstruktur, dessen Form im Ereignistyp definiert ist und die durch die Werte der im Ereignistyp definierten Attribute beschrieben wird.

Das Ereignismodell erstreckt sich über verschiedene Phasen des Leistungsanalysezyklusses. Innerhalb jeder Phase ist eine unterschiedliche Software-Infrastruktur notwendig, um das Ereignismodell beschreiben zu können, doch die Kernaussagen der Modellierung bleiben erhalten. Es ist wichtig, dass Informationen beim Transfer in die nächste Phase nicht verloren gehen, und eine frühere Phase alle grundlegenden Daten für die späteren Phasen bereitstellt. Die durch die Instrumentierung in einer Spur gespeicherten Daten sind redundant, wenn sie von den weiterführenden Phasen, insbesondere der Analysephase, nicht genutzt werden. Innerhalb von KOJAK existiert zwischen den meisten Ereignissen in EPILOG und EARL eine direkte Entsprechung im Ereignistyp. D.h. sowohl in EPILOG, als auch in EARL, gibt es Ereignistypen, die die gleiche Aussage im Ereignismodell treffen. Die einzigen Ausnahmen sind spezielle Ereignisse, die eine Hilfsaufgabe übernehmen. Wenn diese Unterscheidung notwendig ist, werden während des weiteren Kapitels in EPILOG definierte Ereignistypen durch `EREIGNIS` dargestellt und die Ereignistypen von EARL durch Ereignis.

Die ersten grundlegenden Ereignistypen wurden von Felix Wolf innerhalb seiner Dissertation zum Thema der automatischen Leistungsanalyse auf parallelen Computern mit SMP Knoten [18] definiert. Im folgenden werden nun die wichtigsten Teile eines Ereignismodells für die Kommunikation in parallelen Programmen erläutert.

Darstellung der modellierten Beziehungen

Neben der Modellierung der Regionen und des Datenaustauschs, die im Anschluss erläutert werden, ist es wichtig, innerhalb eines Ereignismodells, die Beziehungen zwischen verschiedenen Ereignissen beschreiben zu können. Für die Beschreibung der in dieser Arbeit benutzten Modelle wird auf das Meta-Modell aufgebaut, dass von Felix Wolf bereits in seiner Dissertation [18] benutzt wurde. Um das Verständnis der Diagramme des weiteren Kapitels zu erhöhen, werden dessen Elemente und Beziehungen im folgenden erklärt.

Regionen werden durch eine ausgefüllte rechteckige Box dargestellt. Um die Wahrnehmung auf eine spezielle in einem Diagramm wichtige Region zu lenken, werden alle umfassenden Regionen, die im Diagramm eine untergeordnete Rolle spielen, nicht dargestellt. An ihrer Stelle wird die Zeitlinie als dünner horizontaler Balken dargestellt. Ereignisse werden durch einen Kreis modelliert. Die Beziehungen zwischen den Ereignissen werden durch gerichtete Verbindungen modelliert. Die Richtung geht dabei zeitlich *immer* im Ereignisstrom zurück. D.h. ein Pfeil eines späteren Ereignisses zeigt immer auf ein früheres Ereignis. Dies erleichtert die Verarbeitung des Ereignisstroms vom Anfang bis zum Ende, da so alle Ereignisse, auf die sich ein aktuell zu verarbeitendes Ereignis beziehen kann, selbst bereits verarbeitet wurden. Eine solche Beziehung eines Ereignisses geht immer von *einem* Ereignis zu *einem* anderen Ereignis. Im Modell ist es somit nicht möglich, dass ein Ereignis die selbe Beziehung zu zwei verschiedenen Ereignissen hat. Ein Ereignis kann allerdings mehrere unterschiedliche Beziehungen zu unterschiedlichen Ereignissen besitzen. Eine Unterscheidung der Beziehungen wird durch den Linienstil ausgedrückt. Ein Ereignistyp kann mehrere dieser Beziehungen haben. Welche Beziehungen es insgesamt gibt, ist im Detail in der Dokumentation zu EARL [19] erläutert. Die für die Modellierung der einseitigen Kommunikation notwendigen Beziehungen werden im folgenden kurz erläutert. Ein *enterptr*-Attribut eines Ereignis zeigt immer auf das ENTER-Ereignis der Region im Aufrufbaum, die das Ereignis beinhaltet bzw. umfasst. Wenn das Attribut den Wert *null* besitzt, ist man im Aufrufbaum beim ENTER-Ereignis des alles-umfassenden Hauptprogramms und somit der Wurzel des Aufrufbaums angekommen. Weiterhin wird in den späteren Modellen das Attribut *sendptr* benutzt, das in einem Punkt-zu-Punkt-Datentransfer die Verbindung vom RECV-Ereignis zum dazugehörigen SEND-Ereignis herstellt. Für die einseitige Kommunikation wird ein ähnliches Attribut definiert. *startptr* verbindet das Ende eines einseitigen Datentransfers mit dem Anfang. *originptr* verbindet das Start-Ereignis eines Get-Datentransfers mit dem Origin-Ereignis innerhalb der initiierenden Get-Region auf dem Ursprungsprozess. Das Origin-Ereignis beschreibt den Ursprung der RMA-Operation.

Zur erhöhten Übersichtlichkeit werden in den Diagrammen der Modelle nur solche Beziehungen dargestellt, die dem Verständnis einer konkreten Situation im Ereignismodell dienen. Beziehungen, die aus einem Diagramm herausführen würden, werden nicht angezeigt, da das Ziel für den Betrachter nicht mehr eindeutig nachvollziehbar ist. Die Unterdrückung der Darstellung einer Beziehung ist somit unabhängig von einer tatsächlich bestehenden Beziehung zu einem anderen Ereignis.

Zusätzlich kann der Datentransfer durch einen durchgehenden Pfeil dargestellt werden. Er bezeichnet nur den Datenfluss und keine Beziehung von Ereignissen untereinander. D.h. er dient dem Verständnis des Modells in der gewählten Darstellung und nicht der Modellierung selbst. Deshalb ist es erlaubt, die Laufrichtung des Pfeils von einem Ereignis zu einem

zukünftigen Ereignis zu wählen.

Regionen zur Modellierung des Kontrollflusses

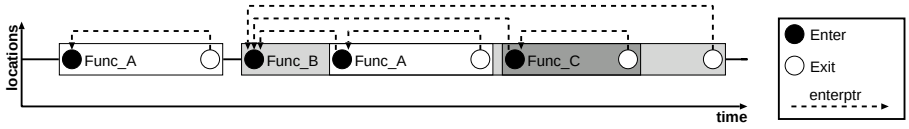
Eines der wichtigsten Elemente innerhalb der Modellierung eines Programmablaufs ist die Lokalisierung des Kontrollflusses im Programm. Dazu werden so genannte Regionen definiert, die folgende Eigenschaften besitzen:

- Eine Region beschreibt eine Anzahl von aufeinander folgenden Instruktionen.
- Regionen können ineinander geschachtelt sein.
- Es gibt keine teilweisen Überschneidungen von Regionen im Ereignisstrom, d.h. eine innere Region muss vor der äußeren Region verlassen werden.
- Die alle anderen umfassende Region ist das Programm selbst.
- Rekursive Aufrufe werden nicht durch ein weiteres Eintrittsereignis modelliert.

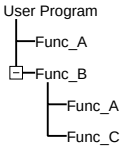
Durch diese Eigenschaften ergibt sich als Struktur ein zirkelfreier Graph. Dieser wird *Aufrufbaum* (calltree) genannt. Der Weg von der Wurzel zu einem bestimmten Zweig heißt *Aufrufpfad* (callpath). Der Aufrufpfad ist eines der Hilfsmittel der Leistungsanalyse, um Leistungsprobleme nicht nur einer Region, sondern einer speziellen Instanz einer Region oder einer Gruppe von Instanzen einer Region zuzuordnen. Der Aufrufpfad gibt dabei zusätzliche Informationen über die Umgebung, in der ein Leistungsproblem besteht. Wie sich aus Regionen eines Ereignisstroms ein Aufrufbaum erstellen lässt, ist in Abbildung 4.1 dargestellt. Der Teil eines Ereignisstroms eines Benutzerprogramms zeigt erst einen Aufruf der Funktion `Func_A` worauf der Kontrollfluss wieder zum Benutzerprogramm zurückkehrt. Dann einen Aufruf von Funktion `Func_B` und einen in diesem Aufruf gekapselten Aufruf von `Func_A`, der nicht zum Benutzerprogramm selbst, sondern zur Funktion `Func_B` zurückkehrt. Innerhalb des gleichen Aufrufs von `Func_B` wird noch eine weitere Funktion, `Func_C` aufgerufen, bevor der Kontrollfluss wieder komplett zum Benutzerprogramm zurückkehrt. Daraus ergibt sich der Aufrufbaum aus Abbildung 4.1(b), der unterhalb des Benutzerprogramms die Funktionen `Func_A` und `Func_B` zeigt, wobei `Func_B` wiederum untergeordnete Aufrufe von `Func_A` und `Func_C` beinhaltet. In der Abbildung 4.1(a) ist im Modell zu erkennen, dass die *enterptr*-Attribute der ENTER-Ereignisse der untergeordneten Funktionsaufrufe auf das ENTER-Ereignis der sie umfassenden Funktion zeigen.

Zur Modellierung von Regionen werden zwei Ereignistypen definiert. Das ENTER-Ereignis bezeichnet den Eintritt in eine Region und das EXIT-Ereignis bezeichnet das Verlassen einer Region. Viele Regionen in einem Programm haben keine besonderen Parameter, die Einfluss auf die Leistungsanalyse nehmen. Deshalb ist es ausreichend, wenn für die Leistungsanalyse nur der reine Eintritt und Austritt mit diesen Ereignistypen festgehalten wird. Funktionen bilden einen Spezialfall einer Region. Sie werden bei der Programmierung genutzt, um mehrfach benötigte Instruktionsfolgen zu kapseln. Sie beschreiben verschiedene Aufgaben, die innerhalb eines Programms bearbeitet werden sollen. Daraus ergibt sich, dass Funktionen auch für die Leistungsanalyse relevante Regionen bilden. Viele Werkzeuge zur Leistungsanalyse bieten deshalb eine automatische Instrumentierung und spätere Aufschlüsselung der gesammelten Leistungsdaten für jede vom Programm genutzte Funktion. Das Laufzeitverhalten einer Funktion ist oft stark von den übergebenen Parametern abhängig. Deshalb kann es wichtig sein, diese Parameter ebenfalls zu speichern. Im verwendeten Modell wird dies über spezielle Austrittsereignisse gewährleistet. D.h. jeder Eintritt in eine Region des Benutzerprogramms wird mit dem gleichen ENTER-Ereignistyp modelliert und abhängig von der Zugehörigkeit der Region zu einer bestimmten Gruppe von Regionen, werden beim Austritt spezielle Ereignistypen verwendet. Diese speziellen Ereignistypen bieten dann zusätzliche Attribute zum Speichern der relevanten Parameter der Funktion.

Die Funktionen der kollektiven Kommunikation in MPI bilden eine solche Gruppe von Regionen. Der Austritt aus einer kollektiven Kommunikationsfunktion wird mit einem MPICEXIT-Ereignis (mpi collective exit) modelliert. Als Attribute werden Kommunikator, Wurzel des Aufrufs und die Anzahl der gesendeten sowie empfangenen Bytes gespeichert. Zusätzlich ist durch dieses Ereignis ausgedrückt, dass es sich um einen kollektiven Aufruf handelt und somit die einzelnen prozesslokalen Regionen prozessübergreifend in Verbindung stehen. Zur Modellierung des kollektiven Austritts aus einer OpenMP-Region ist durch das OMPCEXIT-Ereignis (openmp collective exit) definiert. Es speichert keine zusätzlichen Attribute im Vergleich zum normalen EXIT-Ereignis, drückt durch seinen eigenen Typ jedoch eine prozessübergreifende Zusammengehörigkeit aus. Innerhalb der Erweiterung des Ereignismodells sind weitere Exit-Ereignistypen für die einseitige Kommunikation nötig, die später beschrieben werden.



(a) Regionen innerhalb des Ereignismodells



(b) Resultierender Aufrufbaum

Abbildung 4.1: Modellierung von Regionen und resultierender Aufrufbaum

Modellierung des Nachrichtenaustauschs

Durch das Konzept des Nachrichtenaustauschs selbst, liegt ein großer Fokus der Leistungsanalyse auf dem Senden und Empfangen von Nachrichten. In MPI 1.2 gibt es zwei große Kommunikationsparadigmen. Zum einen die Punkt-zu-Punkt-Kommunikation, bei der zwei einzelne Prozesse Daten austauschen. Zum anderen die kollektive Kommunikation, bei der alle Prozesse eines angegebenen Kommunikators an dem Datenaustausch teilnehmen. Funktionen der kollektiven Kommunikation erzeugen keine weiteren Ereignisse, als die ENTER- und MPICEXIT-Ereignisse ihrer Regionen. Die Werte des Datentransfers, wie zum Beispiel die Anzahl der übertragenen Bytes, werden durch das spezielle MPICEXIT-Ereignis bereits gespeichert. Durch die interne Synchronisation und Kommunikation ist der Zeitpunkt der Datenankunft mit dem Zeitpunkt des MPICEXIT-Ereignisses ausreichend beschrieben. Die zusätzliche Darstellung der ausgetauschten Nachrichten im Ereignisstrom bringt keinen weiteren Informationsgewinn.

Der Punkt-zu-Punkt-Datentransfer wird durch spezielle Ereignisse SEND und RECV beschrieben. Dies ist nötig, da der Datentransfer zwischen zwei Prozessen nicht wie in der kollektiven Kommunikation implizit synchronisiert ist. Der Aufruf des Senders kann zeitlich verschieden vom Aufruf des Empfängers sein, insbesondere bei den nicht-blockierenden Funktionen. Die zusätzlichen Ereignisse zum Datentransfer geben an, dass Daten geschickt bzw. empfangen wurden. Diese werden von den Wrappern der MPI-Funktionen zum Punkt-zu-Punkt-Datentransfer erzeugt. Sie enthalten unter anderem Daten über Sender/Empfänger und Anzahl der gesendeten Bytes. Es kann also auf ein spezielles Austritts-Ereignis verzichtet und für die Modellierung der Region ein EXIT-Ereignis genutzt werden.

In Bezug auf die an einem Datentransfer beteiligten Kommunikationspartner kann die

einseitige Kommunikation in MPI auch als „einseitige Punkt-zu-Punkt-Kommunikation“ angesehen werden. Dies bedeutet, dass an einem Datentransfer immer nur zwei Prozesse gleichzeitig beteiligt sind. Um das Verständnis für die spätere Modellierung dieses Datenaustauschs zu erleichtern, wird im folgenden Kapitel ein Teil des bisherigen Ereignismodells für die Punkt-zu-Punkt-Kommunikation erläutert.

MPI erlaubt Datentransfers durch blockierende oder nicht-blockierende Aufrufe. Bei den blockierenden Aufrufen wird der Kontrollfluss erst an das Anwenderprogramm zurückgegeben, wenn die Übertragungspuffer nicht mehr von der Kommunikationsfunktion benötigt werden. Dies bedeutet, dass ein Sendepuffer wieder verändert und ein Empfangspuffer wieder gelesen werden darf, ohne zu inkonsistenten Prozessdaten zu führen. Bei den nicht-blockierenden Aufrufen wird der Kontrollfluss so schnell wie möglich an das Anwenderprogramm zurückgegeben. Bei der Punkt-zu-Punkt-Kommunikation wird eine *Anforderung* (request) erzeugt, über die der Anwender Informationen über den Status der Datenübertragung erfahren kann. Der Datentransfer wird dann aus Sicht des Anwenders quasi-unabhängig vom Anwenderprogramm abgewickelt. Im Allgemeinen ist der Datentransfer zum Zeitpunkt der Rückgabe des Kontrollflusses noch nicht abgeschlossen und es muss durch Aufrufe anderer MPI Funktionen auf Abgeschlossenheit geprüft werden. Ein nicht-blockierender Datentransfer der Punkt-zu-Punkt-Kommunikation gilt erst als abgeschlossen, wenn die Anforderung korrekt in einer der Funktionen der Wait[1]- oder Test[2]-Funktionsgruppen abgeschlossen wurde. Diese Verlagerung des Abschlusses des Datentransfers in eine andere Region macht eine Verfolgung dieser Anforderungen über Regionsgrenzen hinaus unabdingbar.

MPI kennt als zwei grundsätzliche Anforderungstypen persistente und nicht-persistente Anforderungen. Persistente Anforderungen werden durch eine spezielle Funktion vor dem eigentlichen Datentransfer explizit vom Anwender erzeugt. Nach dem Abschluss des Datentransfers wird diese nicht gelöscht, sondern kann unverändert erneut innerhalb einer nicht-blockierenden Punkt-zu-Punkt-Kommunikation genutzt werden. Dies ist sehr nützlich, wenn sich die Kommunikationsparameter selten ändern. Die Zeit für das Anlegen der Datenstruktur, die Initialisierung der Kommunikationsparameter und für die sonst anfallende Freigabe des Speichers nach erfolgreichem Abschluss des Datentransfers wird gespart. Wird eine nicht-blockierende Punkt-zu-Punkt-Kommunikationsfunktion mit einem nicht vorher initialisierten Anforderungshandle übergeben, so wird eine neue nicht-persistente Anforderungsstruktur erzeugt und das MPI Laufzeitsystem löscht diese Anforderungsstruktur automatisch wieder, nachdem die zugehörige Kommunikation abgeschlossen ist.

Nachträglich können persistente und nicht-persistente Anforderungen nicht vom Benutzerprogramm unterschieden werden. Eine Verfolgung der Kommunikationsanforderungen innerhalb der Instrumentierung beinhaltet somit die korrekte Markierung der Anforderungen als persistent und nicht-persistent. Gleichzeitig dazu unterscheidet MPI die Anforderungen in Sende- und Empfangsanforderungen. Der Versand einer Nachricht startet selbstständig, und der Benutzer kann über die genannten Funktionen Wait und Test erfahren ob der Sendeprozess abgeschlossen ist. MPI benötigt für den Abschluss selbst allerdings diesen Funktionsaufruf nicht. Sendeanforderungen werden selbstständig abgeschlossen, allerdings bleiben die Anforderungen im System. D.h., die für die Anforderung notwendige Datenstruktur wird nicht erneut benutzt, bis die Anforderung zum Beispiel durch `MPI_Request_free` freigegeben wurde.

Im Ereignismodell reicht es aus, den Anfang des Datentransfers an der Quelle der Übertragung und die vollständige Ankunft der Daten am Ziel der Übertragung aufzuzeichnen. Da Sendeanforderungen selbstständig den Datentransfer beginnen, ist dieses Ereignis unabhängig von der Bearbeitung der Anforderungen in Wait oder Test. Für die Leistungsanalyse

[1] `MPI_Wait`, `MPI_Waitany`, `MPI_Waitsome` oder `MPI_Waitall`
[2] `MPI_Test`, `MPI_Testany`, `MPI_Testsome` oder `MPI_Testall`

sind diese Anforderungen deshalb uninteressant und können unbeachtet gelassen werden.

Bei den Empfangsanforderungen ist ein Abschluss durch Wait oder Test jedoch vorgeschrieben, solange der Datentransfer nicht abgebrochen wird. Dies bedeutet eine Abhängigkeit des Abschlussereignisses der Datenübertragung mit dem Aufruf einer dieser Funktionen. Gleichzeitig ist durch die Semantik der Aufrufe eindeutig geregelt, dass das Abschlussereignis unabhängig von der Empfangsfunktion `MPI_Irecv` ist. Die Abschlussereignisse müssen deshalb nicht in der Empfangsfunktion, sondern, wie in Abbildung 4.2 dargestellt, in der Abschlussfunktion behandelt werden. Dort muss auch insbesondere die Persistenz der Anforderung für ihre Behandlung berücksichtigt werden. Im linken Teil des Diagramms ist ein blockierender Datentransfer angegeben, und das Empfangsereignis (Receive) befindet sich innerhalb der Region der Empfangsfunktion `MPI_Recv`. Im rechten Teil der Abbildung ist ein nicht-blockierender Nachrichtenaustausch gezeigt, und nun ist das Empfangsereignis von der Empfangsfunktion `MPI_Irecv` unabhängig und wird im abschließenden Aufruf von `MPI_Wait` modelliert. Es wird somit versucht, das für den Benutzer sichtbare Verhalten zu modellieren.

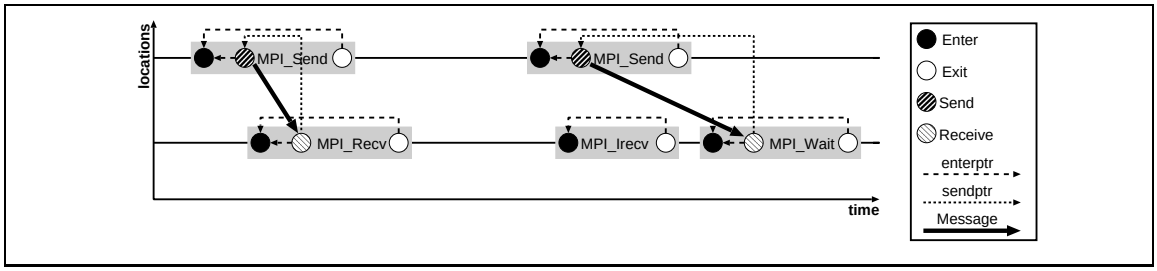


Abbildung 4.2: Modellierung der blockierenden und nicht-blockierenden P2P-Kommunikation

Die nicht-blockierende Kommunikation stellt an das Ereignismodell eine spezielle Anforderung. Ereignisse, die auf logischer Ebene miteinander verknüpft sind, können im Ereignisstrom als unabhängig erscheinen. Bei der Visualisierung der Datenflüsse sind so die Verbindungen nicht mehr, wie von den Namen der Funktionen vermutet, von der Sendefunktion zur Empfangsfunktion dargestellt. Leistungsprobleme sind ebenfalls nicht mehr von dieser Empfangsfunktion abhängig, sondern von dem abschließenden Wait oder Test. Konzepte nicht-blockierender Kommunikation sind deshalb immer mit zusätzlichem Aufwand zu verfolgen und benötigen oft zusätzliche Datenstrukturen, um dies zu ermöglichen. Da die einseitige Kommunikation in MPI als nicht-blockierend implementiert sein kann und die Semantik der Aufrufe einen Abschluss des Datentransfers über die Synchronisationsfunktionen vorschreibt, sind bei der Verfolgung des Datenflusses der RMA-Operationen ebenso weitere Hilfsdatenstrukturen nötig.

4.3 Ereignistypen der einseitigen Kommunikation

Bei der Punkt-zu-Punkt-Kommunikation in MPI gibt es durch die Aufrufe von `MPI_Send` und `MPI_Recv` immer einen expliziten Sender und einen expliziten Empfänger. Der Prozess, der das SEND-Ereignis erzeugt, wird dabei implizit als Initiator des Datentransfers angesehen. Der Empfänger kann den Datentransfer nur vorbereiten, aber nicht starten. Die Quelle der Daten kann damit auch als Ursprung der Datenoperation gesehen werden oder umgekehrt. Bei der einseitigen Kommunikation können diese Aspekte des Datentransfers voneinander getrennt sein. Der Ursprung einer RMA-Operation wird durch den Prozess gebildet, der den Aufruf tätigt. Im Falle einer Get-Operation ist die Quelle der Daten allerdings der Zielprozess der RMA-Operation. Der Begriff des Ziels kann hierbei verwirrend wirken, da die englischen Fachbegriffe *target* und *destination* beide durch Ziel ausgedrückt werden. Das Attribut *target* bezeichnet das Ziel der RMA-Operation und somit das „Gegenüber“ des

Ursprungs der RMA-Operation. Das Attribut *destination* bezeichnet den Bestimmungsort der geschickten Daten. Bei einer Get-Operation ist der Bestimmungsort der geschickten Daten somit der Ursprung der RMA-Operation selbst.

Für die Modellierung dieser Eigenschaften mussten neue Ereignistypen definiert werden, da die genannten Ereignistypen SEND und RECV diese Unterscheidung nicht zulassen. Der Datentransfer der einseitigen Kommunikation wird mit Ereignissen zum Transfer-Start und Transfer-Ende beschrieben. Um eine Unterscheidung zwischen den Datentransfers einer Put-Operation und einer Get-Operation treffen zu können, kann dies in einem Attribut der Transferereignistypen gespeichert werden. Im Falle des hier verwendeten Ereignismodells wurde allerdings dieses Vorgehen zugunsten der Implementation verschiedener Ereignistypen für Put- und Get-Operationen verworfen. Programme mit starker Interaktion der Prozesse erzeugen im dynamischen Programmablauf bei der Verwendung nachrichtenbasierter Kommunikationsparadigmen naturgemäß viele Nachrichten, die im Ereignisstrom modelliert werden müssen. Die Platzersparnis innerhalb des Ereignisstroms durch den Wegfall eines Attributs wiegt hierbei schwerer, als die erhöhte Komplexität des Ereignismodells durch die zusätzlichen Ereignistypen.

Der Datentransfer einer Put-Operation oder Accumulate-Operation wird durch die Ereignistypen PUT_1TS und PUT_1TE modelliert. Die Abkürzung 1TS steht dabei für „one-sided transfer start“, und die Abkürzung 1TE steht für „one-sided transfer end“. Der Datentransfer einer Get-Operation wird analog mit den Ereignistypen GET_1TS und GET_1TE modelliert. Während der Messphase können Daten zum Programmablauf nur prozesslokal gesammelt werden. Ein Austausch der gesammelten Daten noch zur Laufzeit des Programms würde zu starke Störungen im dynamischen Programmablauf hervorrufen. In der Regel sind bei einem Nachrichtenaustausch zwei verschiedene Prozesse beteiligt, von denen allerdings nur der Ursprungsprozess die RMA-Operation startet und somit die Aufrufparameter kennt. Der Zielprozess der RMA-Operation kann deshalb zur Laufzeit keine Ereignisse zum Datentransfer generieren. Daher müssen die entsprechenden Ereignisse im Ereignisstrom des Ursprungs der RMA-Operation festgehalten werden. Um eine spätere Unterscheidung zu gewährleisten, wird das Konzept der *entfernten Ereignisse* (remote events) eingeführt. Dazu werden zwei neue Ereignistypen geschaffen, `ELG_MPI_GET_1TS_REMOTE` und `ELG_MPI_PUT_1TE_REMOTE`, welche während der Messphase dafür genutzt werden, den Start bzw. das Ende eines Datentransfers auf einem entfernten Prozess zu markieren. Diese Ereignisse werden nur während der Messphase benötigt und während der Zusammenführung der Einzelspuren zu einer globalen Ereignisspur durch die entsprechenden lokalen Ereignisse ersetzt. Nach der Umsetzung ist das Ereignis nicht mehr von einem Ereignis zu unterscheiden, welches von Anfang an ein lokales Ereignis beschrieben hat. Deshalb haben diese Ereignistypen keine Entsprechung innerhalb des Ereignismodells oder der Ereignistypen von EARL. Für das Ereignismodell relevant sind nur die in der Abbildung 5.4 *nicht kursiv* gedruckten Ereignistypen der Hierarchie. Die restlichen Ereignistypen der Abbildung gehen aus der spezifischen Implementation des Ereignismodells in EARL hervor und werden in Kapitel 5 näher erklärt. Für das Ereignismodell wird vorausgesetzt, dass ein Ereignistyp alle Attribute des in der dargestellten Hierarchie allgemeineren Ereignistyps ebenfalls besitzt.

In dem erweiterten Ereignismodell, welches in Kapitel 4.4.2 beschrieben ist, kommt es bei der Modellierung des Datentransfers einer Get-Operation zu einer Trennung der modellierten Region der Get-Operation und des veranlassten Datentransfers. Um dies zu beheben wird das GET_1TO-Ereignis (one-sided transfer origin) definiert und im Ereignisstrom zeitlich kurz vor dem entfernten Ereignis des Transfer-Starts gespeichert. Das GET_1TS-Ereignis besitzt das Zeigerattribut *originptr*, welches das GET_1TO Ereignis referenziert. Das GET_1TO-Ereignis stellt dann mit dem eigenen *enterptr*-Attribut die Verbindung zur Region der Get-Operation her. Alle Ereignistypen, die bei der einseitigen Kommunikation

den Datentransfer beschreiben, enthalten das Attribut *rmaid*, welches eine lokal für einen Prozess fortlaufende Nummer für jede RMA-Operation darstellt. Durch dieses Attribut und den Ort des Ereignisses kann eine RMA-Operation eindeutig identifiziert werden.

Bei der einseitigen Kommunikation beziehen sich Aufrufinstanzen immer auf ein spezielles Fenster. Jeder Ereignistyp der einseitigen Kommunikation besitzt ein Attribut *win*, welches das Speicherfenster identifiziert, auf das sich in einem Ereignis bezogen wird.

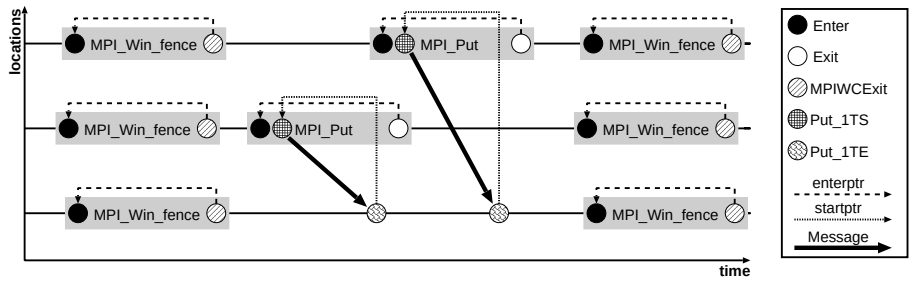
Die Speicherung der Fensteridentifizierung in einem entsprechenden EXIT-Ereignis erzwang die Definition des neuen MPIWEXIT-Ereignistyps. Dadurch werden Regionen der einseitigen Kommunikation identifiziert. Genauer werden durch dieses MPIWEXIT-Ereignis die Regionen der MPI-Funktionen zur allgemeinen Synchronisation mit aktivem Ziel abgeschlossen. Die Regionen der RMA-Operationen `MPI_Put`, `MPI_Get` und `MPI_Accumulate` können durch ein EXIT-Ereignis abgeschlossen werden, da alle Informationen, die die einseitige Kommunikation betreffen bereits in die Ereignisse zur Modellierung des Datentransfers eingegangen sind. Für die später besprochene Modellierung der Synchronisation mit passivem Ziel gilt dies analog. Für die verbleibende Gruppe an Funktionen der einseitigen Kommunikation gilt, dass sie kollektiv aufgerufen werden. `MPI_Win_create`, `MPI_Win_free` und `MPI_Win_fence` benötigen analog zu MPICEXIT das eigene MPIWCEXIT-Ereignis, um die kollektive Natur ihres Aufrufs in die Modellierung aufzunehmen. Die Aufrufe der allgemeinen Synchronisation mit aktivem Ziel beziehen sich immer auf eine MPI-Gruppe von Prozessen. Das MPIWEXIT-Ereignis enthält zur Identifizierung der entsprechend zugehörigen MPI-Gruppe das Attribut *group*. Das MPIWCEXIT-Ereignis enthält analog zusätzlich noch das Attribut *com*, um den benutzten Kommunikator zu identifizieren.

Die Synchronisation mit passivem Ziel arbeitet über Sperren, um den Zugriff auf ein einzelnes Speicherfenster zu koordinieren. Dazu wird der Zeitpunkt, an der ein Prozess eine Sperre anfordert, durch das WLOCK-Ereignis modelliert. Der Ort der gesetzten Sperre wird durch das Attribut *lloc* gespeichert. In dem Attribut *type* wird gespeichert, ob es sich um eine exklusive oder eine gemeinsame Sperre handelt. WUNLOCK bezeichnet das Lösen einer vorher gesetzten Sperre. Es wird ebenfalls der Ort der Sperre im Attribut *lloc* gespeichert, wodurch, in Verbindung mit dem *win*-Attribut aller einseitigen Ereignistypen, eine eindeutige Identifizierung des Speicherfensters ermöglicht wird.

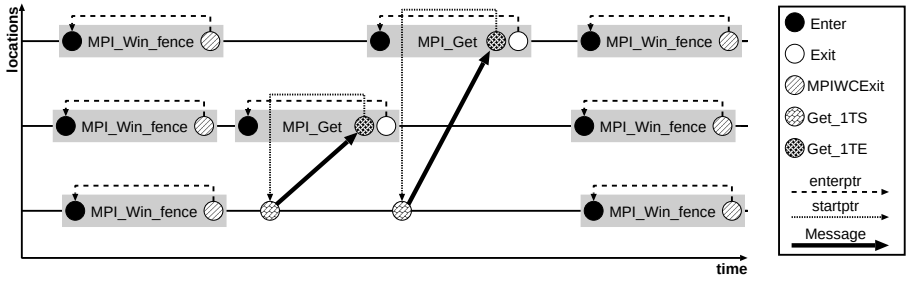
4.4 Entwicklung der Ereignismodelle

Wie bereits beschrieben, sind die wichtigsten Hilfsmittel der ereignisbasierten Leistungsanalyse in der Modellierung des Programmablaufs Regionen und im Falle von MPI auch die Ereignisse des Nachrichtenaustauschs. Mit den erarbeiteten Grundlagen werden nun Modelle beschrieben, die sich in die bestehenden Ereignismodelle von KOJAK nahtlos einfügen lassen. Die für die Funktionen der einseitigen Kommunikation zu modellierenden Regionen müssen innerhalb der Modelle in Zusammenhang gebracht werden. Dieser Zusammenhang zwischen mehreren Regionen und dem Nachrichtenaustausch ist im MPI-Standard beschrieben und wird im weiteren Verlauf nochmals in seinen Auswirkungen auf das konkrete Modell erläutert. Die beiden entwickelten Modelle beschreiben eine Repräsentation der Ereignisse in der aufgezeichneten Spur selbst. Das Basismodell bietet dabei eine einfachere aber ungenauere Modellierung, wohingegen das erweiterte Modell, durch Anpassung der aufgezeichneten Daten, diese Ungenauigkeiten zu minimieren versucht.

Im Kapitel 7 wird noch weiterführend auf die Aspekte eines Ereignismodells eingegangen, die für eine gute Visualisierung des dynamischen Ablaufs des Anwenderprogramms zusätzliche Beachtung finden sollten. Weitere Beziehungen zwischen den Ereignissen können implizit aus dem eingelesenen Ereignisstrom berechnet oder auch durch Definition und Nutzung zusätzlicher neuer Ereignistypen explizit beschrieben werden.



(a) Put-Operation



(b) Get-Operation

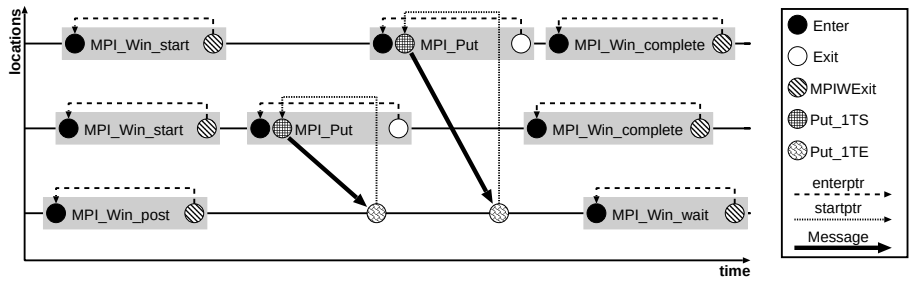
Abbildung 4.3: Synchronisation mit Fence im Basismodell

KOJAK selbst nimmt zur Analyse keine Visualisierung des Ereignisstroms vor. Es ermöglicht lediglich die Umsetzung des Ereignisstroms in ein Format, das von VAMPIR verarbeitet werden kann. Die für die Visualisierung und dortige manuelle Analyse hilfreichen Modellierungen stellen für KOJAK während der Analyse und Suche nach Instanzen von Leistungsproblemen keinen Informationsgewinn dar. Innerhalb der Analyse können alle bisher benötigten Informationen, die nicht explizit durch den Ereignisstrom gegeben sind, berechnet werden.

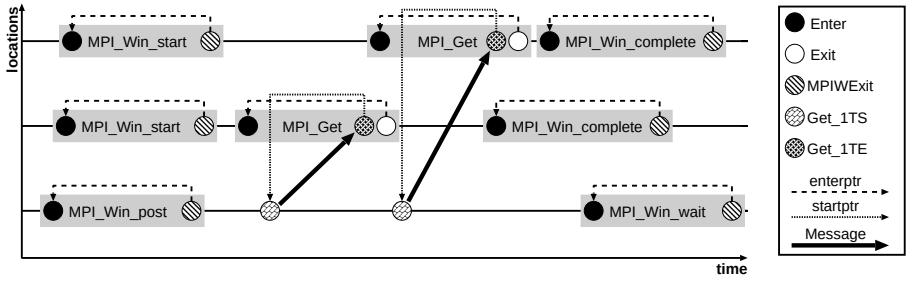
4.4.1 Das Basismodell

Das Basismodell bietet eine grundlegende Sicht auf die dynamischen Abläufe der einseitigen Kommunikation und wird in ähnlicher Form ebenfalls von der Aufzeichnungsbibliothek VAMPIRTRACE benutzt. Bei Eintritt in eine Region der einseitigen Kommunikation wird ein ENTER-Ereignis erzeugt. Bei Austritt aus der Region können verschiedene EXIT-Ereignisse erzeugt werden. RMA-Operationen erzeugen ein normales EXIT-Ereignis, da alle benötigten Informationen über den Datentransfer innerhalb anderer Ereignisse gespeichert werden. Die kollektiven Aufrufe `MPI_Win_create`, `MPI_Win_free` und `MPI_Win_fence` erzeugen ein kollektives MPIWCEXIT-Ereignis, welches zusätzlich zum verwendeten Speicherfenster anzeigt, dass es sich um eine kollektive Operation handelt. Die Funktionen der allgemeinen Synchronisation mit aktivem Ziel erzeugen ein spezielles MPIWEXIT-Ereignis, das ebenfalls die Identifizierung des verwendeten Speicherfensters ermöglicht und zusätzlich noch die Gruppe an Prozessen speichert, mit denen der lokale Prozess synchronisieren muss. Diese Gruppe ist im Allgemeinen eine Untergruppe der gesamten im Kommunikator der Speicherfensterdefinition enthaltenen Prozesse. Bei der Synchronisation mit passivem Ziel werden die synchronisationsbezogenen Daten, wie bei den RMA-Operationen, in eigenen Ereignissen gespeichert, sodass diese Regionen mit einem EXIT-Ereignis verlassen werden.

Bei der Modellierung des Datentransfers werden die ENTER- und EXIT-Ereignisse der aufgerufenen RMA-Operation als Grenzen für den Datentransfer herangezogen. D.h. im Modell ist der Datentransfer mit der Rückkehr des Kontrollflusses zum Anwenderprogramm abgeschlossen. In Kapitel 3 wurde darauf hingewiesen, dass RMA-Operationen im allgemeinen



(a) Put-Operation



(b) Get-Operation

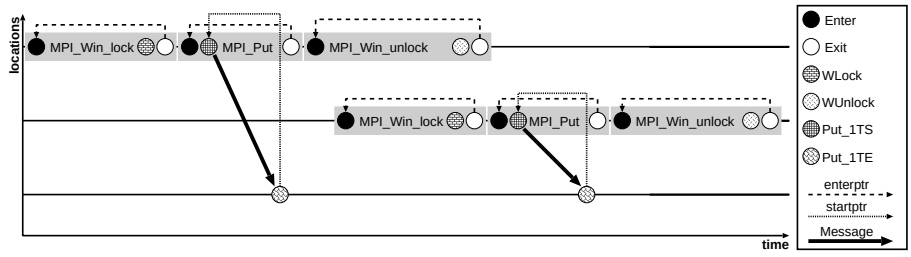
Abbildung 4.4: Allgemeine Synchronisation mit aktivem Ziel im Basismodell

nicht-blockierend implementiert werden. Das bedeutet, dass der Kontrollfluss vorzeitig zum Anwenderprogramm zurückkehrt. Damit ist die Modellierung des Datentransferabschlusses auf die Rückkehr des Kontrollflusses im allgemeinen nicht realitätsgetreu. Allerdings bietet dieses Modell eine möglichst einfache Weise der Abbildung und erfordert von der Implementierung der Erstellung der globalen Spur keinerlei zeitbezogene Bearbeitung. Die Abbildungen 4.3, 4.4 und 4.5 beschreiben die Modellierung der einzelnen Ereignisse im Ereignisstrom.

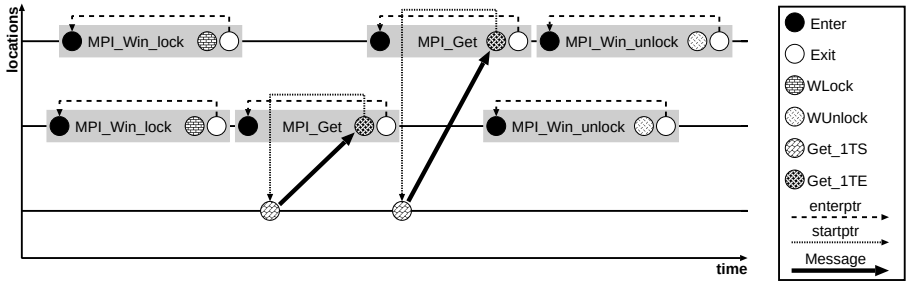
Das Ereignis des startenden Datentransfers ist zeitlich kurz hinter dem ENTER-Ereignis der RMA-Operation modelliert. Damit wird ausgedrückt, dass der Datentransfer so schnell wie möglich beginnt, unabhängig von weiteren Ereignissen. Das Ende des Datentransfers ist zeitlich kurz vor dem Austritt aus der RMA-Operation modelliert. Eine Verbindung des Datentransfers mit der Region der RMA-Operation ist gegeben, da entweder das Start-Ereignis oder das End-Ereignis innerhalb der entsprechenden Region liegt und durch das *enterptr*-Attribut mit den ENTER-Ereignis der Region verbunden ist.

Eine Anforderung einer Sperre der einseitigen Kommunikation wird durch das WLOCK-Ereignis modelliert, welches direkt vor dem EXIT-Ereignis der Region des `MPI_Win_lock`-Aufrufs im Ereignisstrom aufgezeichnet wird. Das Aufheben der Sperre wird analog direkt vor dem EXIT-Ereignis durch ein WUNLOCK gekennzeichnet.

Die bereits angesprochene Bibliothek SHMEM ermöglicht durch Hardware-Unterstützung einen einseitigen Datentransfer, der bei der Rückkehr des Kontrollflusses zum Anwenderprogramm als abgeschlossen angenommen werden kann. Für diesen Fall ist das Ereignismodell für den Datentransfer zwar korrekt, da die Synchronisation für den Datentransfer auf unterster Ebene nicht benötigt wird, aber die Semantik der einseitigen Kommunikation in MPI ist dadurch nicht vollständig ausgedrückt. Das Erfassen der Synchronisation im Ereignisstrom ist weiterhin wichtig, da einige Leistungsmerkmale mittels Ereignissen der Synchronisation der RMA-Operation definiert werden können.



(a) Put-Operation mit exklusiver Sperre



(b) Get-Operation mit gemeinsamer Sperre

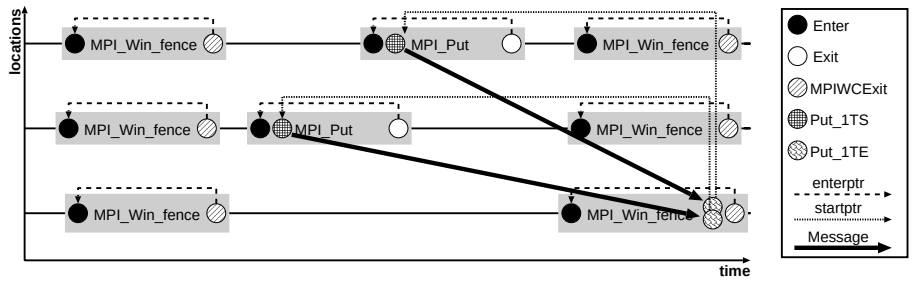
Abbildung 4.5: Synchronisation mit passivem Ziel im Basismodell

Vor- und Nachteile

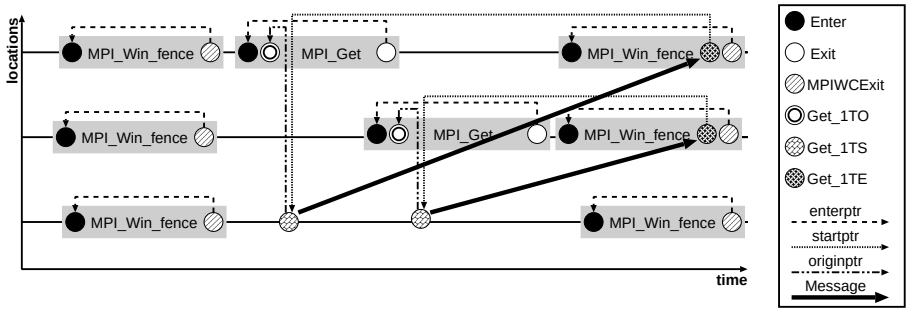
Kernpunkt der Kritik des Basismodells ist, dass der Datentransfer, aufgrund der Modellierung zwischen den ENTER- und EXIT-Ereignissen, unabhängig von der Synchronisation zu sein scheint. In der Realität ist dies allerdings nie gegeben. Auch bei einer Implementation mit Hilfe der beschriebenen Bibliothek SHMEM, ist der Datentransfer zwar höchstwahrscheinlich abgeschlossen, doch die umschließenden Synchronisationsaufrufe vor dem nächsten Zugriff sind weiterhin nötig, um standardkonform auf die übertragenen Daten zuzugreifen. Dadurch können Wartezeiten im Programmablauf entstehen, die in der Visualisierung für den Anwender nicht direkt mit dem Datentransfer zusammenhängen, da dieser als abgeschlossen dargestellt wird. Gleichzeitig können Aussagen über Bandbreiten irreführend sein, da im Falle einer nicht-blockierenden Implementation einer RMA-Operation erheblich höhere Bandbreiten errechnet werden, als tatsächlich vorliegen. Zusätzlich sollten Werte wie Bandbreiten inklusive eines Protokolloverheads ausgerechnet werden. Dies geht nur unter Berücksichtigung der umfassenden Synchronisation.

Das Basismodell bietet allerdings auch Vorteile. Es stellt eine Abbildungskompatibilität zum verwendeten Ereignismodell der einseitigen Kommunikation in der weit verbreiteten Analyseumgebung VAMPIR her. Anwender haben so die Möglichkeit mit KOJAK ein Ereignismodell zu nutzen, welches eine sehr ähnliche Abbildung der Ereignisse zu VAMPIR erzeugt, aber zum Beispiel im Bereich der allgemeinen Synchronisation mit aktivem Ziel zumindest die Darstellung der Regionen der Synchronisation hinzufügt. Abbildung 4.9 auf Seite 48 zeigt diese Situation noch einmal genauer. Die Funktionsaufrufe der allgemeinen Synchronisation mit aktivem Ziel, werden durch VAMPIRTRACE in der Version 4 noch nicht aufgezeichnet. Dadurch scheint bei der Visualisierung (Abb. 4.9(a)) vor und nach der RMA-Operation mehr Zeit durch die Anwendung selbst verbraucht zu werden, die in der Realität allerdings von MPI benötigt wird. Dies wird in der Darstellung der im Basismodell durch EPILOG aufgezeichneten Spur in Abbildung 4.9(b) deutlich.

Außerdem ist es während der Messphase sehr leicht, die Ereignisse an der Stelle aufzuzeichnen und in den Ereignisstrom einzufügen, an der sie modelliert werden sollen. Dies vereinfacht eine Implementierung dieser Art der Aufzeichnung erheblich.



(a) Put-Operation



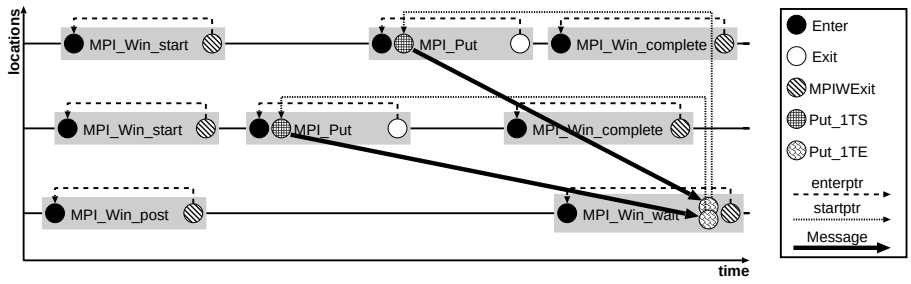
(b) Get-Operation

Abbildung 4.6: Synchronisation mit Fence im erweiterten Modell

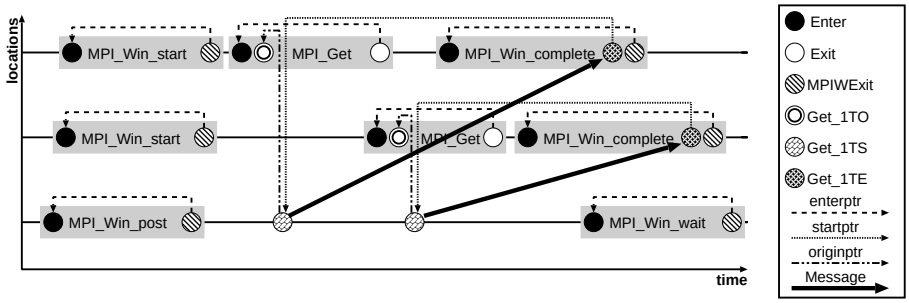
4.4.2 Das erweiterte Modell

Um die Nachteile des Basismodells zu überwinden, ist es nötig, ein Modell zu schaffen, welches die Ereignisse sowohl standardkonform als auch realitätsnah abbilden kann. Mit den in Kapitel 3.5 definierten Regeln, welche im weiteren Verlauf durch R1-R6 referenziert werden, lassen sich Zeitpunkte ableiten, zu denen eine RMA-Operation auf einem Prozess spätestens abgeschlossen ist. Die Abgeschlossenheit einer RMA-Operation bestimmt im größten Maße ihre Leistung. Eine Übertragung mit hoher Bandbreite nützt wenig, wenn der Abschluss der gesamten Operation verzögert getätigt wird. Das vorhandene Modell muss dementsprechend erweitert werden. Dies geschieht durch eine Modellierung der Ereignisse im Ereignisstrom anhand der durch den MPI-Standard vorgegebenen Regeln. Die Abbildungen 4.6, 4.7 und 4.8 stellen die aus dieser Verschiebung resultierenden neuen Ereignis-Zeitpunkte relativ zu ihren umgebenden Ereignissen, wie ENTER- und EXIT-Ereignisse, schematisch dar und werden nachfolgend beschrieben.

Die Start-Ereignisse eines Datentransfers verbleiben am Beginn der RMA-Operation. Die End-Ereignisse eines Datentransfers werden am Ende der zugehörigen Synchronisationsfunktion abgebildet. Dadurch kann es, wie in Abbildung 4.6(a) dazu kommen, dass zwei Ereignisse quasi-gleichzeitig modelliert werden. Wie schon in der Einführung zu den Ereignistypen angesprochen, wird in diesem Modell so eine Trennung der Region einer Get-Operation von dem Datentransfer, den die Operation veranlasst, hervorgerufen. Um erneut eine Verbindung zu schaffen, wird innerhalb der Region der Get-Operation ein zusätzliches Ereignis des Typs GET_1TO modelliert, welches über das eigene *enterptr*-Attribut die Beziehung zur Region herstellt. Das GET_1TS-Ereignis auf dem entfernten Prozess kann nun dieses Ereignis mittels des eigenen *originptr*-Attributs mit dem modellierten Datentransfer verbinden.



(a) Put-Operation



(b) Get-Operation

Abbildung 4.7: Allgemeine Synchronisation mit aktivem Ziel im erweiterten Modell

Synchronisation mit Fence

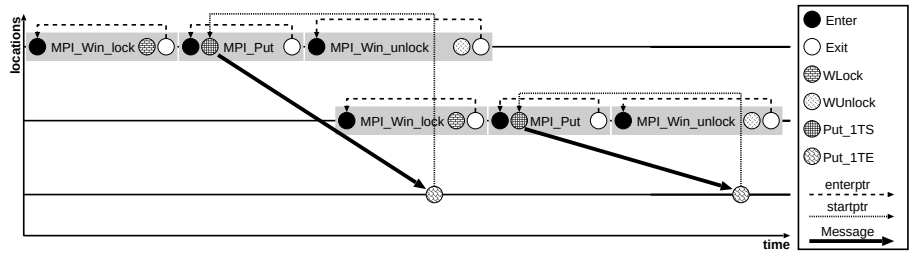
R1 besagt, dass eine RMA-Operation am Ursprung des Aufrufs abgeschlossen ist, sobald diese aus dem synchronisierenden Aufruf von `MPI_Win_fence` zurückkehrt. Im Falle eines `MPI_Put` oder `MPI_Accumulate` wird am Ursprung der RMA-Operation der Beginn des Datentransfers durch ein PUT_1TS markiert. Bei der RMA-Operation `MPI_Get` wird auf dem Zielprozess der Get-Operation der Beginn des Datentransfers durch GET_1TS modelliert. Get-Operationen besitzen noch das zusätzliche Ereignis GET_1TO welches den Ursprung der RMA-Operation markiert. Das Ende des Datentransfers wird aufgrund der Regel R1 nun direkt vor dem MPIWCEXIT-Ereignis modelliert. In Abbildung 4.6 ist dies im Modell dargestellt.

Die Abgeschlossenheit einer RMA-Operation auf dem Zielprozess wird durch R2 bestimmt. Dort wird angegeben, dass der späteste Zeitpunkt, an dem ein RMA-Aufruf auf dem Ziel als abgeschlossen gilt, mit der Rückkehr des Kontrollflusses aus dem korrespondierenden Fence-Aufruf zusammenfällt. Die Modellierung des Datentransfer-Endes wird hier analog direkt vor das MPIWCEXIT-Ereignis auf dem Zielprozess der RMA-Operation modelliert.

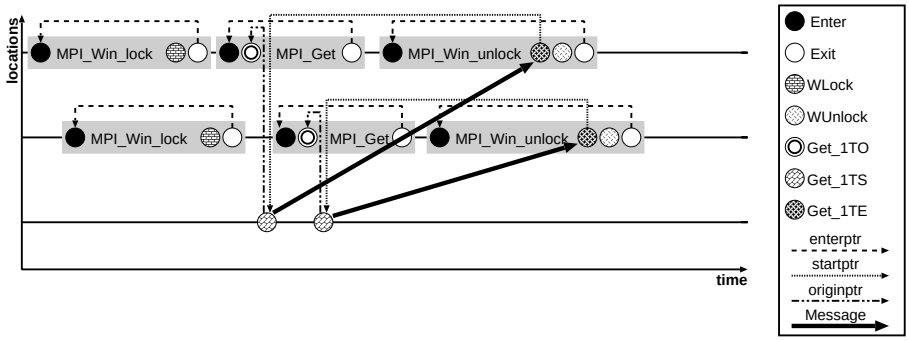
R6 gibt dazu noch an, dass die Änderung der öffentlichen Kopie des Fensters lokal spätestens dann beim Ursprung oder Ziel sichtbar wird, wenn der Ursprung oder das Ziel den Fence-Aufruf verlassen. Das bedeutet, dass mit dem Austritt aus dem Fence-Aufruf auf beiden Seiten auch die Änderungen lokal verfügbar sind. Dies unterstützt die beschriebene Modellierung.

Allgemeine Synchronisation mit aktivem Ziel

Bei der allgemeinen Synchronisation mit aktivem Ziel folgt, wie bei einer Synchronisation mit Fence, aus R1, dass am Ursprung eines RMA-Aufrufs der Datentransfer abgeschlossen ist, wenn der Kontrollfluss nach der zugehörigen Synchronisation zurückkehrt. Im Falle der allgemeinen Synchronisation ist dies der Aufruf von `MPI_Win_complete`. PUT_1TS- und GET_1TS-Ereignisse werden deshalb auch in diesem Teil des Modells direkt nach dem



(a) Put-Operation mit exklusiver Sperre



(b) Get-Operation mit gemeinsamer Sperre

Abbildung 4.8: Synchronisation mit passivem Ziel im erweiterten Modell

ENTER-Ereignis in der Spur markiert. Das GET_1TE-Ereignis wird direkt vor dem MPIWEXIT der Region des Complete-Aufrufs modelliert.

Auf dem Zielprozess des RMA-Aufrufs werden RMA-Operationen durch einen Aufruf von `MPI_Win_wait` oder `MPI_Win_test` mit positivem Flag abgeschlossen. R3 definiert den Zeitpunkt der Rückkehr aus diesen Aufrufinstanzen als spätesten Zeitpunkt des Abschlusses. Deshalb werden PUT_1TE-Ereignisse auf dem Zielprozess des RMA-Aufrufs direkt vor dem MPIWEXIT des entsprechenden Aufrufs von `MPI_Win_wait` oder `MPI_Win_test` modelliert.

In Abbildung 4.7 werden diese Zusammenhänge noch einmal verdeutlicht.

Synchronisation mit passivem Ziel

Regel R4 gibt an, dass eine RMA-Operation sowohl am Ursprung, als auch am Ziel mit dem Verlassen des Unlock-Aufrufs abgeschlossen wird. Dies bedeutet, dass das GET_1TE-Ereignis direkt vor dem EXIT-Ereignis des `MPI_Win_unlock`-Aufrufs modelliert wird. Das Ereignis PUT_1TE wird zeitlich ebenfalls direkt vor dem EXIT-Ereignis am RMA-Ursprung modelliert, allerdings als Ereignis des Zielprozesses der RMA-Operation. Wie beim Basismodell kann dies auf dem Zielprozess der RMA-Operation innerhalb einer beliebigen Region liegen.

Analog zu der Modellierung der Sperren im Basismodell werden die Ereignisse WLOCK und WUNLOCK jeweils direkt vor dem EXIT-Ereignis erzeugt, welche die Region des Aufrufs von `MPI_Win_lock` bzw. `MPI_Win_unlock` abschließt. Abbildung 4.8 stellt die Modellierung der Ereignisse der Synchronisation mit passivem Ziel noch einmal bildlich dar.

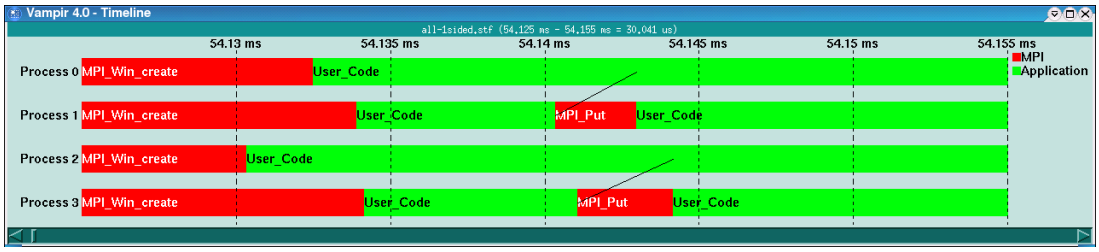
Vor- und Nachteile

Durch die Berücksichtigung der Angaben des MPI-Standards bei der Modellierung der Ereignisse des Datentransfers ergibt sich nun ein Abbild der Ereignisse, welches semantisch

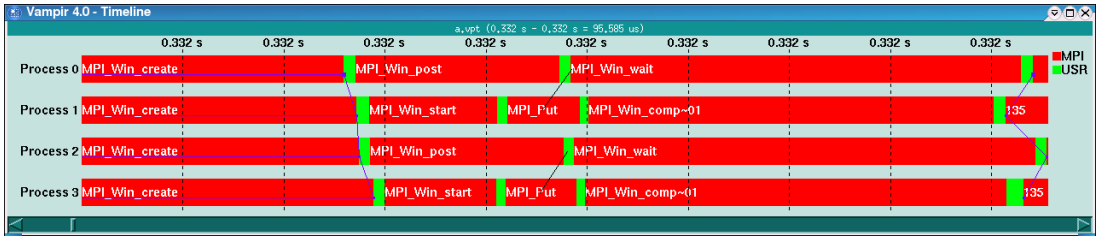
korrekt, standardkonform und in vielen Fällen auch realitätsnah ist. Durch die Modellierung der Abschlüsse der Datentransfers an den spätmöglichsten Zeitpunkt wird eine obere Grenze für die Datenankunft geschaffen. Das Basismodell bietet dazu die untere Grenze. Der Datentransfer wird in der Realität zwischen diesen Grenzen liegen. Mit der im erweiterten Modell getätigten Annahme über die Ankunft der Daten, kann der Anwender leichter abschätzen, ob lokale Zugriffe zeitlich im richtigen Rahmen liegen (siehe dazu Abb. 4.9(c) im Vergleich zu Abb. 4.9(b)).

Aus Anfang und Ende der Datenübertragung kann eine Übertragungsbandbreite ermittelt werden, die den Overhead einer Synchronisation mit einbezieht. D.h. der Anwender kann davon ausgehen, dass er mindestens die errechnete Bandbreite bei der Datenübertragung erreicht hat.

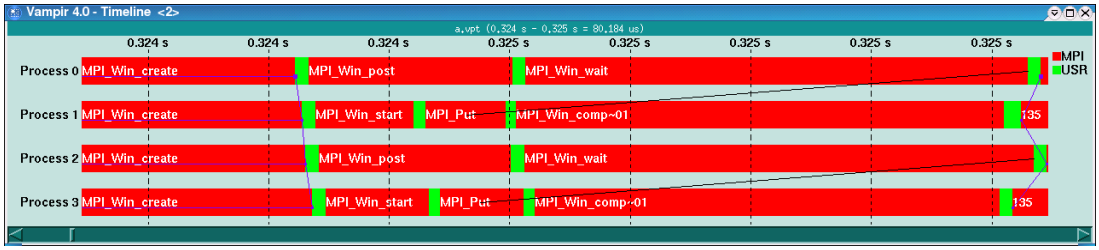
Der Nachteil dieses Ereignismodells ist eine komplexere Implementierung, welche den Speicherverbrauch und die Rechenzeit bei der Zusammenführung der Einzelspuren erhöht. Aufgrund der Tatsache, dass bisher noch keine großen Anwendungen zur Analyse bereitstanden, kann eine Erhöhung dieser Faktoren nicht genau abgeschätzt werden. Gleichzeitig bewirkt die Modellierung eine visuelle Trennung des Datentransfers von der Region der RMA-Operation bei dem Aufruf eines `MPI_Get`. Dies ist für die von KOJAK durchgeführte Analyse durch das `GET_1TO`-Ereignis korrigiert, allerdings wird die manuelle Analyse mit Hilfe der Visualisierung des Ereignisstroms für den Anwender erschwert. In Kapitel 7 wird genauer auf diese Aspekte eingegangen.



(a) Ausschnitt einer VAMPIRTRACE-Aufzeichnung



(b) Ausschnitt einer EPILOG-Aufzeichnung im Basismodell



(c) Ausschnitt einer EPILOG-Aufzeichnung im erweiterten Modell

Abbildung 4.9: Aufzeichnung der allgemeinen Synchronisation mit aktivem Ziel

4.4.3 Modellierungsaspekte der Sperren bei den vorgestellten Modellen

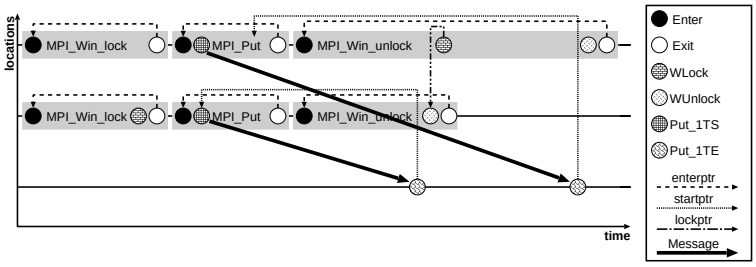
Die Modellierung der Sperren bei der Synchronisation mit passivem Ziel ergibt Schwierigkeiten, die mit den bisherigen Mitteln nur unzufriedenstellend gelöst werden können. Der MPI-Standard erlaubt explizit, sowohl `MPI_Win_lock`, als auch `MPI_Put` nicht-blockierend zu implementieren, wodurch die komplette Handhabung der Sperrzuweisung des Datentransfers und der Sperraufhebung von der Funktion `MPI_Win_unlock` getätigt wird.

Mit WUNLOCK kann direkt das Lösen einer Sperre modelliert werden, dahingegen stellt LOCK nur die Anforderung einer Sperre dar. Dies ist unterschiedlich zum Lock-Unlock-Modell, welches bei der Modellierung der OpenMP-Konstrukte genutzt wird. Da es durch den MPI-Standard erlaubt ist, die komplette Operation innerhalb des Aufrufs von `MPI_Win_unlock` abzuwickeln, kann zur Messzeit kein geeigneter Zeitpunkt plattformübergreifend und korrekt modelliert werden, an dem die Zuweisung einer Sperre erfolgt.

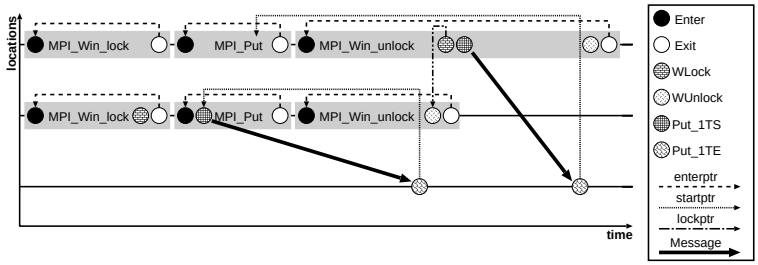
Ein Verbindungsattribut, wie es durch SYNC.*lockptr* beschrieben wird, welches alle zu einem Sperrobjekt zugehörigen Ereignisse zu einer Kette verbindet, ist für die einseitige Kommunikation nicht praktikabel. Deshalb ist die Verbindung zwischen den WLOCK- und WUNLOCK-Ereignissen im momentanen Modell nicht möglich. Durch die fehlende Verbindung ist es erschwert, Leistungsmerkmale zu definieren. Eine mögliche Lösung der Situation soll im folgenden diskutiert werden.

Wie später im Kapitel 5 zur Implementierung gezeigt wird, wird das erweiterte Ereignismodell im Ereignisstrom erst nach der Messphase erstellt. Dazu werden Verschiebungen einzelner Ereignisse im Ereignisstrom vorgenommen. Dabei kann sich die prozesslokale Reihenfolge der Ereignisse ändern. Eine analoge Verschiebung der aufgezeichneten WLOCK-Ereignisse könnte an dieser Stelle eine zeitliche alternierende Reihenfolge von WLOCK- und WUNLOCK-Ereignissen in der Spur erzeugen, die dann durch ein entsprechendes Zeigerattribut verbunden werden können. Der Zeitpunkt, an den das Ereignis verschoben werden müsste ist in einem Szenario in Abbildung 4.10(a) angegeben. Durch diese Verschiebung impliziert, ist die zusätzliche Verschiebung des Beginns des Datentransfers auf einen Zeitpunkt *nach* der Zuweisung einer Sperre. Dies ist in Abbildung 4.10(b) veranschaulicht.

Die Stelle an der identifiziert werden kann, welcher Prozess als nächstes die Sperre zugewie-



(a) Verschiebung von WLOCK



(b) Verschiebung des Starts der RMA-Operation

Abbildung 4.10: Lösungsansatz für die Modellierung der Sperren

sen bekommen hat, liegt im Ereignisstrom aber hinter des zu modellierenden Zeitpunkts. Eine Pufferung aller Ereignisse bis zu diesem Zeitpunkt ist nicht praktikabel, da dies eine unbestimmte Menge an Ereignissen aufnehmen müsste und somit unter Umständen sehr viel Speicherresourcen verbraucht.

Wie in Kapitel 5 noch beschrieben wird, wird der Ereignisstrom momentan bereits zweimal durchlaufen. Eine Lösung, die eine Verschiebung innerhalb dieser beiden Durchläufe realisiert ist somit wünschenswert. Wenn die zu realisierende Reihenfolge der WLOCK-Ereignisse innerhalb des ersten Durchlaufs ermittelt wird, in dem keinerlei Ereignisse in die Ausgabespur geschrieben werden, und so eine Datenstruktur aufgebaut wird, die im zweiten Durchlauf während des Festschreibens der Ereignisse in der Ausgabespur diese Datenstruktur nutzt, müssen erheblich weniger Ereignisse gepuffert werden.

Eine Lösung dieser Problematik ist für die Behandlung der Synchronisation mit passivem Ziel in der Zukunft wünschenswert, um die Definition von Leistungsmerkmalen zu erleichtern und den damit verbundenen Aufwand zu minimieren.

Kapitel 5

Erweiterung von KOJAK

5.1 Motivation

Um die in Kapitel 4 angeführten Ereignismodelle für die einseitige Kommunikation im Ereignisstrom abzubilden, muss eine Instrumentierung der beteiligten Funktionen bereitgestellt werden. Im folgenden Kapitel wird ein für die Funktionen der einseitigen Kommunikation implementierter Wrapper exemplarisch dargestellt. Teile der Informationen, die diese Wrapper aufzeichnen müssen werden oft innerhalb anderer Wrapper erzeugt oder verändert. Deshalb müssen Datenstrukturen, mit denen die Wrapper während der Ereignisaufzeichnung Daten austauschen können und Funktionen, die den Zugriff auf diese Datenstrukturen ermöglichen, geschaffen werden. Die für die einseitige Kommunikation erweiterten Datentypen und Funktionen werden im weiteren Kapitel beschrieben. Zu den Funktionen der einseitigen Kommunikation aus der MPI-2-Erweiterung mussten auch zusätzlich die MPI-Funktionen zur Gruppenverwaltung aus MPI 1.2 instrumentiert werden. MPI-Gruppen sind kein direktes Konstrukt für die einseitige Kommunikation. Da die allgemeine Synchronisation mit aktivem Ziel allerdings Gruppen für die Zugriffssynchronisation nutzt, müssen Informationen über diese Gruppen für die Ereigniserzeugung bereitgestellt werden. Im weiteren wird auf die Funktionen zur EPILOG-internen Verwaltung dieser Gruppen, der Speicherfenster, den zu modellierenden Datentransfer und die Sperren der einseitigen Kommunikation eingegangen.

Dann wird die Zusammenführung der aufgezeichneten Einzelspuren, insbesondere die Verarbeitung der Ereignisse der einseitigen Kommunikation, erklärt. Dazu wird aufgeführt, welche Änderungen an den aufgezeichneten Daten durchgeführt werden müssen, um zum einen das Basismodell und zum anderen das erweiterte Modell im Gesamtstrom der Ereignisse zu implementieren. Es wird gezeigt, wie die Modelle aufeinander aufbauen. Dazu werden die nötigen Schritte der Bearbeitung der Ereignisse, die in beiden Ereignismodellen gleich sind, anhand des Basismodells erklärt. Darauf aufbauend wird gezeigt, welche zusätzlichen Änderungen nötig sind, um den Ereignisstrom aus dem Basismodell in das erweiterte Modell zu überführen.

Im letzten Teil des Kapitels wird auf die Anfänge der Änderungen und Erweiterungen der EARL-Schnittstelle eingegangen, um die Informationen über die Ereignisse der einseitigen Kommunikation, welche während der Messphase aufgezeichnet werden, für die Analysephase verfügbar zu machen. Dies beinhaltet die Implementation der zusätzlichen Ereignistypen, die in Kapitel 4 bereits angesprochen wurden.

5.2 Änderungen für die Messphase

5.2.1 Wrapper

MPI ist eine Beschreibung von Gruppen von Funktionen zum Nachrichtenaustausch und ihrer Funktionalität. Eine Implementierung stellt diese Funktionalität unter den im MPI-Standard beschriebenen Funktionssymbolen zur Verfügung. Diese Funktionen werden durch Wrapper instrumentiert, um leistungsrelevante Daten vor und nach der Ausführung der Funktionen speichern zu können. Diese Wrapper sind Bestandteil der EPILOG-Bibliothek. Die Wrapper in EPILOG haben im Allgemeinen eine sehr ähnliche Struktur, da von den instrumentierten Funktionen immer ein Eintritts- und ein Austrittsereignis erzeugt werden muss. Dazu werden abhängig von der speziellen Aufgabe der MPI-Funktion noch zusätzliche Ereignisse erzeugt oder andere Funktionen aufgerufen, welche die benötigten Hilfsstrukturen für die Ereignisse bearbeiten. Im folgenden Beispiel 5.1 ist der Wrapper für `MPI_Put` exemplarisch dargestellt.

```
int MPI_Put(void *origin_addr, int origin_count,
            MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count,
            MPI_Datatype target_datatype, MPI_Win win)
{
  int result, sendsz, recvsz;
  elg_ui4 dpid;
  elg_ui4 wid;

  if (IS_TRACE_ON)
    {
      TRACE_OFF();
#ifndef ELG_CSITE_INST
      elg_enter(elg_mpi_regid[ELG__MPI_PUT]);
#endif
      dpid = elg_win_rank_to_pe( target_rank, win );
      wid = elg_win_id( win );

      PMPI_Type_size(origin_datatype, &sendsz);
      elg_mpi_put_1ts(dpid, wid, ELG_NEXT_RMA_ID,
                      origin_count*sendsz);
      result = PMPI_Put( origin_addr, origin_count,
                    origin_datatype, target_rank, target_disp,
                    target_count, target_datatype, win );

      PMPI_Type_size(target_datatype, &recvsz);
      elg_mpi_put_1te_remote(dpid, wid, ELG_CURR_RMA_ID,
                             target_count*recvsz);
#ifndef ELG_CSITE_INST
      elg_exit();
#endif
      TRACE_ON();
    }
  else
      result = PMPI_Put( origin_addr, origin_count,
                    origin_datatype, target_rank, target_disp,
                    target_count, target_datatype, win );
  return result;
}
```

Beispiel 5.1: Wrapper für `MPI_Put`

Durch die Abfrage auf IS_TRACE_ON in Zeile 10 wird ermittelt, ob vor dem Aufruf der Funktion die Erzeugung der Ereignisse aktiviert war oder nicht. Wenn für diesen Aufruf des Wrappers die Erzeugung von Ereignissen aktiviert war, so wird diese deaktiviert, bevor der Wrapper mit der Aufzeichnung fortfährt. Dies bewirkt, dass eventuelle Aufrufe von vorinstrumentierten Funktionen innerhalb von EPILOG nun keine Ereignisse erzeugen. Dies könnte z.B. geschehen, wenn eine MPI-Bibliothek zur Implementierung der einseitigen Kommunikation auf Funktionen der Punkt-zu-Punkt-Kommunikation zurückgreift bzw. umgekehrt. Danach kann mit der eigentlichen Ereigniserzeugung durch die Instrumentierung fortgefahren werden.

Wenn die Erzeugung nicht aktiviert war, wird nur die MPI-Funktion selbst mit den übergebenen Parametern aufgerufen und der von diesem Aufruf zurückgegebene Wert wird an die Funktion weitergereicht, die diesen Wrapper aufgerufen hat. Die Abfrage auf IS_TRACE_ON enthält jeder Wrapper in EPILOG, und verhindert so, dass innerhalb einer instrumentierten Funktion andere instrumentierte Funktionen Ereignisse aufzeichnen.

Jeder Wrapper in EPILOG zeichnet mindestens die ELG_ENTER- und ELG_EXIT-Ereignisse auf, wenn dies nicht durch eine so genannte Callsite-Instrumentierung durch den Compiler automatisch übernommen wird. Der Anwender muss dies zur Übersetzungszeit der EPILOG-Bibliothek angeben, da die Entscheidung, ob EPILOG eigene Kontrollflussereignisse erzeugt, durch den C-Präprozessor getätigt wird, wie hier in den Zeilen 13-15 und 29-31.

Bis zu diesem Punkt ist die Struktur jedes Wrappers gleich. Jetzt folgen Instruktionen, die auf die explizite Aufgabe der Funktion zugeschnitten sind. Um das nächste Ereignis aufzuzeichnen, werden noch weitere Informationen benötigt. In Zeile 16 wird durch elg_win_rank_to_pe anhand des übergebenen Fensters der globale Rang des Prozesses, dessen lokaler Rang in target_rank übergeben wurde, ermittelt. Die Ränge können sich unterscheiden, da target_rank den Rang des Prozesses relativ zum Kommunikator angibt, mit dem das Speicherfenster definiert wurde. Wenn also das Speicherfenster nicht mit dem Kommunikator MPI_COMM_WORLD definiert wurde, wird der hier ermittelte Rang im Allgemeinen unterschiedlich zum übergebenen Rang sein. Es ist wichtig in den Ereignissen die globalen Ränge zu speichern, da dies nach Beendigung des Programms nicht mehr zufriedenstellend gelöst werden kann.

In der darauf folgenden Zeile 17 wird der Bezeichner des Speicherfensters ermittelt, um in Zeile 20 das ELG_MPI_PUT_1TS-Ereignis für den Beginn des Datentransfers zu erzeugen. Für das Attribut der übertragenen Bytes muss noch mittels der MPI-Funktion PMPI_Type_size die Größe der zur Übertragung genutzten Datenstruktur in Bytes ermittelt werden. Direkt nach der Erzeugung des ELG_MPI_PUT_1TS-Ereignisses erfolgt der Aufruf der Put-Operation über die PMPI-Schnittstelle. Wenn der Aufruf zum Wrapper zurückkehrt, kann der Abschluss begonnen werden. Dazu wird in diesem Wrapper noch das ELG_MPI_PUT_1TE_REMOTE-Ereignis erzeugt, um den Abschluss des Datentransfers auf dem Zielprozess des RMA-Aufrufs zu kennzeichnen. Der Modellierungszeitpunkt entspricht dem Basismodell. Direkt nach der Erzeugung des Abschluss-Ereignisses des Datentransfers in Zeile 27 wird in Zeile 29, sofern keine Callsite-Instrumentierung durch den Compiler vorliegt, das ELG_EXIT-Ereignis erzeugt. In Zeile 38 wird der Rückgabewert der Funktionsaufrufe aus Zeile 22 bzw. 34 an die aufrufende Instanz zurückgeliefert.

5.2.2 Fensterverwaltung

Eines der wichtigsten Bezugsobjekte der einseitigen Kommunikation ist das Speicherfenster auf dem operiert wird. Innerhalb von MPI wird für die Referenzierung dieser Speicherfenster ein Handle des Typs MPI_Win benutzt. Wie dieser Datentyp beschaffen ist, d.h. ob es ein einfacher Datentyp oder eine zusammengesetzte Datenstruktur ist, bleibt der Implementa-

tion freigestellt. Um eine portable Aufzeichnung der MPI-Abläufe zu gewährleisten, muss während der Messphase eine geeignete Abstraktion gewählt werden, die unabhängig von der jeweiligen Implementation von MPI auf dem Messsystem ist. Um Speicherplatz zu sparen, wird, wie für ähnliche Objekte der Punkt-zu-Punkt- und kollektiven Kommunikation, ein Definitionseintrag für ein Fenster generiert, der die charakteristischen Daten über das betreffende Fenster enthält. Dazu gehört die eindeutige Identifizierung des Speicherfensters über das Attribut `wid`. Das weitere Attribut `cid` enthält den durch einen `ELG_MPI_COMM`-Eintrag definierten Bezeichner des MPI-Kommunikators, über den dieses Fenster definiert wurde. Dieser Eintrag muss bereits erzeugt sein. Der Aufbau des Definitionseintrags ist in Tabelle 5.1 angegeben.

`ELG_MPI_WIN`		
Datentyp	**Bezeichner**	**Beschreibung**
`elg_ui4`	`wid`	window identifier
`elg_ui4`	`cid`	communicator identifier

Tabelle 5.1: Aufbau des `ELG_MPI_WIN`-Definitionseintrags

Für die Analyse ist es nur wichtig, ein Speicherfenster eindeutig identifizieren zu können. Eventuelle interne Parameter der Erzeugung des Speicherfensters sind nicht nötig. Deshalb wird ein 4-Byte breiter ganzzahliger Bezeichner für ein Speicherfenster definiert. Nach der Aufzeichnung des Programmablaufs wird das Handle für das Speicherfenster nicht mehr genutzt und das Fenster rein über diesen Bezeichner referenziert. Während der Messphase muss somit eine korrekte Umsetzung des Handles auf die intern von EPILOG gewählte Identifikationsnummer bereitgestellt werden. Zur Umsetzung wird eine lineare Liste verwendet, die in ihren Einträgen zum einen das MPI-Handle des Speicherfensters und zum anderen die von EPILOG zugewiesene Identifikationsnummer aufnimmt. Zur Verwaltung der Einträge in dieser Liste wurden die drei Funktionen

- `void elg_win_create(MPI_Win win, MPI_Comm comm)`
- `void elg_win_free(MPI_Win win)`
- `elg_ui4 elg_win_id(MPI_Win win)`

erstellt. Durch einen Aufruf von `elg_win_create` wird ein Definitionseintrag für dieses Fenster im Ereignisstrom erzeugt, anschließend dem Speicherfenster eine neue Identifikationsnummer zugewiesen und diese in die Liste eingetragen. Der Aufruf erfolgt innerhalb der Instrumentierung der Funktion `MPI_Win_create`. Durch `elg_win_free` wird der entsprechende Eintrag aus der Liste entfernt. Die Funktion wird innerhalb der Instrumentierung von `MPI_Win_free` ausgeführt. Mit `elg_win_id` wird zu einem übergebenen MPI-Speicherfenster-Handle die zugewiesene Identifikationsnummer zurückgeliefert. Diese Funktion wird von verschiedenen Funktionen genutzt, die einen Datensatz speichern müssen, in dem der Bezeichner des Speicherfensters benötigt wird.

Die Funktionen der allgemeinen Synchronisation mit aktivem Ziel werden alle durch eine Kombination von ENTER- und MPIWEXIT-Ereignissen modelliert. Als Attribut der MPIWEXIT-Ereignisse wird der Gruppenbezeichner benötigt, der an dem Zugriff auf das Speicherfenster beteiligt war. MPI verwaltet diese Daten für die abschließenden Funktionen `MPI_Win_complete`, `MPI_Win_wait` und `MPI_Win_test` intern, sodass für den Aufruf dieser Funktionen lediglich das Handle für das Speicherfenster übergeben wird. Die Struktur des MPIWEXIT-Ereignisses ist in der folgenden Tabelle 5.2 dargestellt.

Dass trotz der Synchronisation über MPI-Gruppen, in diesem Eintrag durch `cid` ein *communicator*-Bezeichner gespeichert wird, folgt aus der gemeinsamen Abbildung von `MPI_Comm`- und `MPI_Group`-Objekten. Im folgenden Kapitel 5.2.3 werden die Hintergründe dafür aufgezeigt. In der angegebenen Struktur wird zusätzlich das Flag `synex` gespeichert, um zu kennzeichnen, dass mit diesem Ereignis eine RMA-Epoche abgeschlossen wurde. Dies ist

ELG_MPI_WINEXIT		
Datentyp	**Bezeichner**	**Beschreibung**
elg_ui4	lid	location identifier
elg_d8	time	time stamp
elg_ui8 | elg_d8	metv[]	metric values
elg_ui4	wid	window identifier
elg_ui4	cid	communicator identifier
elg_ui1	synex	synchronization exit flag

Tabelle 5.2: Aufbau des ELG_MPI_WINEXIT-Ereigniseintrags

insbesondere wichtig zur Kennzeichnung eines Abschlusses über `MPI_Win_test` und dessen MPIWEXIT-Ereignisses. Um nun zur Erzeugung eines solchen Ereignisses die korrekten Attributwerte zu erhalten, muss eine Verfolgung der RMA-Epochen und eine Zuordnung der beteiligten Gruppen bereitgestellt werden. Um dies zu ermöglichen müssen die Zugriffe bereits zum Zeitpunkt der Funktionen, die eine Epoche starten verfolgt werden. D.h. innerhalb der Instrumentierung der Funktionen `MPI_Win_start` und `MPI_Win_post` muss die Gruppe in einem Format hinterlegt werden, die für die abschließenden Funktionen zugreifbar ist.

Dies wird erneut durch eine lineare Liste realisiert, dessen Einträge diesmal den Speicherfenster-Bezeichner, den Gruppen-Bezeichner sowie ein Zusatzflag aufnehmen. Für die Verwaltung der offenen Fensterepochen während der allgemeinen Synchronisation mit aktivem Ziel wurden die folgenden drei Funktionen implementiert:

- `void elg_winacc_start(MPI_Win win, MPI_Group group, elg_ui1 color)`
- `void elg_winacc_end(MPI_Win win, elg_ui1 color)`
- `elg_ui4 elg_winacc_get_gid(MPI_Win win, elg_ui1 color)`

Ein Aufruf von `elg_winacc_start` wird innerhalb der Instrumentierung der Funktionen `MPI_Win_start` und `MPI_Win_post` aufgerufen, um einen entsprechenden Eintrag in der Liste vorzunehmen. Durch einen Aufruf von `elg_winacc_end` wird in Aufrufen von `MPI_Win_complete`, `MPI_Win_wait` und `MPI_Win_test`, die eine Epoche abschließen, dieser Eintrag aus der Liste entfernt. Ein Aufruf von `elg_winacc_get_gid` ermittelt den Gruppenbezeichner der Gruppe, die gerade eine Epoche auf das übergebene Fenster offen hat. Das Flag `color` zeigt bei den Funktionen mit dem Wert 0 an, dass es sich um eine Freigabeepoche handelt und ein Wert von 1 zeigt an, dass es sich um eine Zugriffsepoche handelt. Dies ist nötig, da, obwohl die gleichen Prozesse insgesamt beteiligt sind, die MPI-Funktionen verschiedene Gruppen erhalten (vgl. Kapitel 3.4.1, Seite 26). In MPI sind RMA-Zugriffe der Prozesse auch auf eigene RMA-Speicherfenster erlaubt. In diesem Fall existieren dann auf dem Prozess gleichzeitig eine Freigabe- und eine Zugriffsepoche für dasselbe Speicherfenster. Durch das Flag wird sichergestellt, dass `elg_winacc_get_gid` die korrekte Gruppe zurückliefern kann.

Bei den kollektiven Funktionen der einseitigen Kommunikation ist eine solche Verfolgung der Zugriffe nicht nötig, da bei der Synchronisation mit Fence immer der gesamte Kommunikator teilnimmt, der das Speicherfenster definiert hat. Die kollektiven Funktionen erzeugen ein ELG_MPI_WINCOLLEXIT-Ereignis, welches den in Tabelle 5.3 angegebenen Aufbau hat.

Als besonderes Attribut dieses fensterbezogenen Austrittsereignisses, reicht es aus, den Bezeichner des Fensters zu speichern, auf das sich dieser Aufruf bezieht. Über diesen Bezeichner kann wiederum der beteiligte Kommunikator erlangt werden und somit die Liste der beteiligten Prozesse.

5.2.3 Gruppenverwaltung

Durch die bisherige Unterstützung der kollektiven Kommunikation durch KOJAK ist bereits eine weitestgehende Verfolgung der Kommunikator-verwaltenden Funktionen gewährleistet. Es existiert ein Kommunikator-Definitionseintrag `MPI_COMM` für eine EPILOG-Spur. Die Struktur des Definitionseintrags ist in der folgenden Tabelle 5.4 angegeben.

`ELG_MPI_COMM`		
Datentyp	**Bezeichner**	**Beschreibung**
`elg_ui4`	`cid`	communicator identifier
`elg_ui1`	`uag`	*used as group* indicator
`elg_ui4`	`grpc`	size of the bit string in bytes
`elg_ui1`	`grpv[grpc]`	bit string defining the group

Tabelle 5.4: Aufbau des `ELG_MPI_COMM`-Definitionseintrags

Die dort verwendete Modellierung orientiert sich an der Betrachtung, dass ein MPI-Kommunikator eine Gruppe von Prozessen mit einem speziellen Kontext ist. Der Kontext ist für die spätere Analyse nicht mehr wichtig, und so bleibt für die Speicherung eine Abbildung der zu diesem Kommunikator beteiligten globalen Prozessränge. Eine möglichst kompakte Art, diese Information zu speichern, liegt in der Darstellung einer Liste aller Prozessränge als Bitvektor. Der Prozess mit Rang i wird dabei durch das $i$te Bit in diesem Bitvektor repräsentiert. Die Anzahl der Bytes `grpc` des Bitvektors `grpv` bestimmt sich aus der Gesamtzahl der MPI-Prozesse.

$$\texttt{grpc} = \left\lceil \frac{\texttt{MPI\_Comm\_size(MPI\_COMM\_WORLD)}}{8} \right\rceil$$

Die durch die Größenanpassung überschüssigen Bits des letzten Bytes werden bei der Nutzung des Bitvektors ignoriert. Diese werden als nicht-gesetzt gespeichert.

Für die allgemeine Synchronisation mit aktivem Ziel wurde bereits erwähnt, dass eine Verfolgung der MPI-Gruppen zusätzlich zu den MPI-Kommunikatoren nötig ist. Da die MPI-Kommunikatoren bereits als Prozessgruppen modelliert wurden, ist es sinnvoll diese Modellierung für die MPI-Gruppen zu erweitern. Dabei kann es interessant sein, zu unterscheiden, ob ein EPILOG-Kommunikator-Objekt während des Programmablaufs als MPI-Gruppe oder als MPI-Kommunikator genutzt wurde. Dazu wird das Flag `uag` genutzt. Es kennzeichnet alle Kommunikator-Objekte, die in einem Gruppenkontext benutzt wurden. Ein Wert von 0 bezeichnet dabei die Nutzung als reiner MPI-Kommunikator und ein Wert von 1 bezeichnet dabei die Nutzung innerhalb eines MPIWEXIT-Ereignisses. Durch diese Verfolgung der Benutzung werden die `ELG_MPI_COMM`-Datensätze während der Zusammenführung der Einzelspuren gesammelt und als letzte Definitionseinträge vor den, den Definitionsteil abschließenden, Einträgen `ELG_NUM_EVENTS` und `ELG_LAST_DEF` in der resultierenden Spur abgelegt.

Um die Abbildung der MPI-Kommunikatoren und MPI-Gruppen auf die EPILOG-Kommunikator-Objekte korrekt zu handhaben, wird wiederum eine Liste verwendet. Diese enthält

`ELG_MPI_WINCOLLEXIT`		
Datentyp	**Bezeichner**	**Beschreibung**
`elg_ui4`	`lid`	location identifier
`elg_d8`	`time`	time stamp
`elg_ui8` | `elg_d8`	`metv[]`	metric values
`elg_ui4`	`wid`	window identifier

Tabelle 5.3: Aufbau des `ELG_MPI_WINCOLLEXIT`-Ereigniseintrags

als Datensätze eine Datenstruktur mit einem `MPI_Comm`-Attribut, ein `MPI_Group`-Attribut und dem EPILOG-internen Bezeichner `cid`. Zur Bearbeitung dieser Liste sind verschiedene Funktionen notwendig.

- `void elg_comm_init()`
- `void elg_comm_finalize()`

Die beiden Funktionen `elg_comm_init` und `elg_comm_finalize` werden in den Wrappern der Funktionen `MPI_Init` und `MPI_Finalize` benutzt. Innerhalb von `elg_comm_init` wird die Anzahl der zur Modellierung des Kommunikators `MPI_COMM_WORLD` benötigten Bytes ermittelt und global gespeichert. Zusätzlich wird der erste Definitionseintrag für `MPI_COMM_WORLD` erzeugt, da dieser Kommunikator implizit während der Initialisierung von MPI erzeugt wird und keinen weiteren Aufruf einer Benutzerfunktion benötigt. Zusätzlich wird die Liste für die Umsetzung mit statischer Größe angelegt. Dies limitiert zwar die möglichen gleichzeitig existierenden MPI-Kommunikatoren und -Gruppen, benötigt bei der Eintragung neuer Objekte allerdings kein aufwendiges Speichermanagement. Die Funktionen zum Hinzufügen und Löschen benötigen so eine besser abzuschätzende Laufzeit. In `elg_comm_finalize` wird der für die Liste und Objekte reservierte Speicher wieder freigegeben.

- `int elg_rank_to_pe(int rank, MPI_Comm comm)`

Die Funktion `elg_rank_to_pe` liefert analog zu der bereits für die Fenster-Verwaltung angeführten Funktion `elg_win_rank_to_pe` den globalen Rang eines Prozesses zu einem übergebenen Kommunikator.

- `void elg_group_to_bitvector(MPI_Group group)`

Die Hilfsfunktion `elg_group_to_bitvector` wird während der Eintragung eines neuen EPILOG-Kommunikator-Objekts benötigt, um die Ränge der beteiligten Prozesse in einem Bitvektor abzubilden. Die Abbildung erfolgt innerhalb einer globalen Hilfsstruktur, die während der Ausführung von `elg_comm_init` angelegt wurde. Deshalb besitzt diese Funktion keine sichtbaren Ausgabeparameter.

- `void elg_comm_create(MPI_Comm comm)`
- `void elg_comm_free(MPI_Comm comm)`
- `elg_ui4 elg_comm_id(MPI_Comm comm)`

Diese drei Funktionen dienen der reinen Listenverwaltung. `elg_comm_create` trägt den übergebenen MPI-Kommunikator in die Liste der aktiven Bezeichner ein. Da noch keine Gruppe dazu definiert wurde, wird als Bezeichner für die zugehörige Gruppe die Konstante `MPI_GROUP_NULL` gespeichert. `elg_comm_free` löscht den zum übergebenen MPI-Kommunikator passenden Eintrag aus der Liste und `elg_comm_id` liefert den verwendeten Bezeichner des übergebenen MPI-Kommunikators.

- `void elg_group_create(MPI_Group group)`
- `void elg_group_free(MPI_Group group)`
- `elg_ui4 elg_group_id(MPI_Group group)`
- `int elg_group_search(MPI_Group group)`

Die letzten vier Verwaltungsfunktionen für Bezeichner der internen Kommunikator-Objekte wurden erstellt, um die bisherige Listenverwaltung durch die Verarbeitung übergebener MPI-Gruppen-Objekte zu unterstützen. `elg_group_create` wird innerhalb eines Wrappers einer MPI-Funktion ausgeführt, die als Rückgabeparameter ein MPI-Group-Objekt besitzt. Dadurch können für neu angelegte MPI-Gruppen direkt die entsprechenden Bezeichner erzeugt und in die Liste der gültigen Bezeichner eingetragen werden. Gibt es für die angelegte Gruppe keinen zugehörigen MPI-Kommunikator wird im Listeneintrag der Wert

MPI_COMM_NULL verwendet. Der Aufruf von elg_group_free innerhalb der MPI-Funktion MPI_Group_free ermöglicht die Freigabe eines Gruppen-Bezeichners. elg_group_id liefert, wie das Pendant elg_comm_id für Kommunikatoren, den Bezeichner für die übergebene MPI-Gruppe zurück. Die Funktion elg_group_search wird während der Erzeugung einer neuen MPI-Gruppe oder eines neuen MPI-Kommunikators genutzt, um in der Liste eventuell eingetragene Gruppen wiederzufinden. Dadurch können Kommunikatoren nachträglich mit der ihnen zugehörigen MPI-Gruppe versehen oder Kommunikatoren zu bestehenden Gruppen erzeugt werden. Da Kommunikatoren in MPI immer eine zugehörige Gruppe besitzen, wird auf diese Weise Speicherplatz gespart.

5.2.4 Datentransfer

Wie in dem Beispiel-Wrapper der MPI_Put-Funktion bereits gezeigt, werden die Ereignisse zur Modellierung des Datentransfers innerhalb der Wrapper der RMA-Operationen selbst erzeugt. MPI_Put und MPI_Accumulate nutzen dabei die Ereignistypen ELG_MPI_PUT_* und MPI_Get nutzt die Ereignistypen ELG_MPI_GET_*. Das verwendete Ereignismodell benutzt insgesamt fünf verschiedene Ereignis-Datensatztypen, um den Datentransfer zu modellieren. Dazu müssen noch zwei weitere Ereignistypen in der Messphase dazugenommen werden, um die Einschränkung der rein prozesslokalen Aufzeichnung zu kompensieren. Diese Einträge entsprechen in Eintragsgröße und Attributen den lokalen Ereignistypen, mit der Ausnahme, dass die Werte von lid und dlid bzw. slid im Vergleich zu den lokalen Ereignissen vertauscht sind.

ELG_MPI_PUT_1TS

Durch diesen Datensatz wird der Beginn einer Put-Operation modelliert. Sein Aufbau ist in Tabelle 5.5 dargestellt. dlid gibt den Rang des Zielprozesses an, an den der Transfer gerichtet ist. wid identifiziert das RMA-Speicherfenster, das bei dem Datentransfer genutzt werden soll. rmaid gibt eine prozesslokal eindeutige Identifikationsnummer für die RMA-Operation an. Durch rmaid und den Rang des Prozesses kann eine RMA-Operation immer eindeutig bestimmt werden. sent gibt die Größe der gesendeten Nachricht in Bytes an.

ELG_MPI_PUT_1TS		
Datentyp	**Bezeichner**	**Beschreibung**
elg_ui4	lid	location identifier
elg_d8	time	time stamp
elg_ui4	dlid	destination location identifier
elg_ui4	wid	window identifier
elg_ui4	rmaid	rma identifier
elg_ui4	sent	message length in bytes

Tabelle 5.5: Aufbau des ELG_MPI_PUT_1TS-Ereigniseintrags

ELG_MPI_PUT_1TE / ELG_MPI_PUT_1TE_REMOTE

Die Ereignisse ELG_MPI_PUT_1TE bzw. ELG_MPI_PUT_1TE_REMOTE kennzeichnen den Abschluss einer durch ELG_MPI_PUT_1TS begonnenen RMA-Operation. Tabelle 5.6 gibt den Aufbau des Ereigniseintrags wider. Der Wert des Attributs recvd entspricht der Größe der empfangenen Nachricht in Bytes. Ein ELG_MPI_PUT_1TE-Datensatz wird kann nur auf dem Zielprozess der Put-Operation vorkommen, wohingegen ELG_MPI_PUT_1TE_REMOTE lediglich am Ursprung der RMA-Operation vorkommen kann. Gleichzeitig kann das entfernte Ereignis nur innerhalb einer Einzelspur auftreten, da diese während der Zusammenführung

der Einzelspuren zu einer Gesamtspur ersetzt werden. Das bedeutet, dass innerhalb *einer* Ereignisspur nicht gleichzeitig Ereignisse der beiden Typen enthalten sein können.

ELG_MPI_PUT_1TE / ELG_MPI_PUT_1TE_REMOTE		
Datentyp	**Bezeichner**	**Beschreibung**
elg_ui4	lid	location identifier
elg_d8	time	time stamp
elg_ui4	slid	source location identifier
elg_ui4	wid	window identifier
elg_ui4	rmaid	rma identifier
elg_ui4	recvd	message length in bytes

Tabelle 5.6: Aufbau des ELG_MPI_PUT_1TE- und ELG_MPI_PUT_1TE_REMOTE-Ereigniseintrags

ELG_MPI_GET_1TO

Dieser Eintrag bezeichnet den Ursprung einer Get-Operation. Er wird, während der Ersetzung des ELG_MPI_GET_1TS_REMOTE-Ereignisses durch das ELG_MPI_GET_1TS-Ereignis, auf dem Ursprungsprozess generiert. Auf diese Weise kann nach Zusammenführung der Spuren eine Verbindung vom Transferbeginn zur Region der Get-Operation einfacher modelliert werden. Es wird für diese Zuordnung lediglich die rmaid benötigt, da die vollständigen Informationen des Datentransfers im ELG_MPI_GET_1TS-Ereignis gespeichert werden. Alle drei Ereignisse der Get-Operation enthalten den gleichen Wert in rmaid. Dadurch kann die Zusammengehörigkeit dieser Ereignisse einfacher erkannt werden. Der Aufbau des Eintrags wird in der folgenden Tabelle 5.7 dargestellt.

ELG_MPI_GET_1TO		
Datentyp	**Bezeichner**	**Beschreibung**
elg_ui4	lid	location identifier
elg_d8	time	time stamp
elg_ui4	rmaid	rma identifier

Tabelle 5.7: Aufbau des ELG_MPI_GET_1TO-Ereigniseintrags

ELG_MPI_GET_1TS / ELG_MPI_GET_1TS_REMOTE

Diese Ereignistypen beschreiben den Beginn des Datentransfers einer Get-Operation der einseitigen Kommunikation. Das entfernte Ereignis ELG_MPI_GET_1TS_REMOTE wird während der Messphase aufgezeichnet und während der Zusammenführung der Einzelspuren in das entsprechende lokale Ereignis ELG_MPI_GET_1TS umgewandelt. Der Aufbau der Ereignistypen ist in Tabelle 5.8 angegeben. Weiterhin gelten die gleichen Aussagen über die Attribute wid, rmaid und sent, wie sie schon bei der Put-Operation genannt wurden.

ELG_MPI_GET_1TS / ELG_MPI_GET_1TS_REMOTE		
Datentyp	**Bezeichner**	**Beschreibung**
elg_ui4	lid	location identifier
elg_d8	time	time stamp
elg_ui4	dlid	destination location identifier
elg_ui4	wid	window identifier
elg_ui4	rmaid	rma identifier
elg_ui4	sent	message length in bytes

Tabelle 5.8: Aufbau des ELG_MPI_GET_1TS- und ELG_MPI_GET_1TS_REMOTE-Ereigniseintrags

ELG_MPI_GET_1TE

Dieses Ereignis, dessen Aufbau in Tabelle 5.9 angegeben ist, kennzeichnet das Ende eines durch `MPI_Get` realisierten Datentransfers. `slid` gibt die Quelle der Daten an. Das Attribut `wid` identifiziert das Speicherfenster, von dem Daten übertragen wurden. Das Attribut `rmaid` dient zur erleichterten Identifizierung aller zusammenhängenden Teile der RMA-Operation. Dazu gibt `recvd` die Größe der empfangenen Daten in Bytes an.

ELG_MPI_GET_1TE		
Datentyp	**Bezeichner**	**Beschreibung**
elg_ui4	lid	location identifier
elg_d8	time	time stamp
elg_ui4	slid	source location identifier
elg_ui4	wid	window identifier
elg_ui4	rmaid	rma identifier
elg_ui4	recvd	message length in bytes

Tabelle 5.9: Aufbau des ELG_MPI_GET_1TE-Ereigniseintrags

5.2.5 Sperren

Die Modellierung der Sperren der Synchronisation mit passivem Ziel wird durch die beiden Ereignistypen ELG_MPI_WIN_LOCK und ELG_MPI_WIN_UNLOCK abgedeckt. Durch dieses Ereignispaar wird die maximale Breite einer Zugriffsepoche während der Synchronisation mit passivem Ziel beschrieben.

ELG_MPI_WIN_LOCK

Durch dieses Ereignis wird der Zeitpunkt modelliert, an dem ein Prozess innerhalb der Synchronisation mit passivem Ziel eine Sperre angefordert hat. Das Fenster, auf das die gewünschte Sperre gesetzt werden soll, ist durch den Bezeichner `wid` gekennzeichnet. Da es möglich ist, dass der Aufruf von `MPI_Win_lock` nicht blockiert, ist dies nicht zwangsläufig der Zeitpunkt, an dem dieser Prozess die Sperre auch zugesprochen bekommt. Dieses Ereignis markiert somit nur den frühest möglichen Zeitpunkt, an dem der Prozess die Sperre zugesprochen bekommen haben kann. Durch das Attribut `ltype` wird die Art der Sperre angegeben. Es kann den Wert 0 für eine gemeinsame Sperre und den Wert 1 für eine exklusive Sperre enthalten. Der Aufbau des Ereigniseintrages ist in der folgenden Tabelle 5.10 aufgeführt.

ELG_MPI_WIN_LOCK		
Datentyp	**Bezeichner**	**Beschreibung**
elg_ui4	lid	location identifier
elg_d8	time	time stamp
elg_ui4	llid	lock location identifier
elg_ui4	wid	window identifier
elg_ui1	ltype	lock type

Tabelle 5.10: Aufbau des ELG_MPI_WIN_LOCK-Ereigniseintrags

ELG_MPI_WIN_UNLOCK

Da in den Regeln zum korrekten Abschluss von RMA-Operationen das Verlassen der Funktion `MPI_Win_unlock` als Abschluss einer RMA-Operation sowohl auf dem Ursprungsprozess als auch auf dem Zielprozess der Operation gilt, modelliert das Ereignis

ELG_MPI_WIN_UNLOCK die Rückgabe der Sperre an das MPI-System. Der reale Datentransfer ist zum Zeitpunkt der Generierung definitiv abgeschlossen, da dieses Ereignis direkt nach dem eigentlichen Aufruf der Unlock-Funktion innerhalb des Wrappers generiert wird. Das angegebene Fenster auf dem durch `llid` identifizierten Prozess kann somit als entsperrt angenommen werden. Der Aufbau des Ereigniseintrages kann der Tabelle 5.11 entnommen werden.

ELG_MPI_WIN_UNLOCK		
Datentyp	**Bezeichner**	**Beschreibung**
elg_ui4	lid	location identifier
elg_d8	time	time stamp
elg_ui4	llid	lock location identifier
elg_ui4	wid	window identifier

Tabelle 5.11: Aufbau des ELG_MPI_WIN_UNLOCK-Ereigniseintrags

5.3 Zusammenführung der Einzelspuren

Während der Laufzeit des Programms wird versucht, das Programm selbst so gering wie möglich in seinem Ablauf zu stören. Ein komplexes Ereignismodell zur Aufzeichnungszeit erfüllt diese Anforderung meist nicht. Um Suchvorgänge innerhalb instrumentierungseigener Datenstrukturen möglichst zu vermeiden, wird während der Messphase ein Ereignis ohne Zwischenspeicherung in Warteschlangen direkt im Ereignisstrom gespeichert. Die dafür genutzten Zeitpunkte zur Modellierung der Ereignisse entsprechen dem Basismodell. Dieses Modell wird während der Messphase benutzt, unabhängig davon, welches Modell für die spätere globale Spur verwendet werden soll.

Im folgenden Kapitel werden die für die Zusammenführung der Einzelspuren nötigen Schritte erläutert. Dabei wird erst auf die grundlegende Verarbeitung der Spureinträge mit ihrem doppelten Durchlauf durch die vorhandenen Ereignisströme eingegangen. Danach werden im Unterkapitel über das erweiterte Modell die zur Umsetzung der Ereignisse auf spätere Zeitpunkte nötigen Funktionen erläutert.

5.3.1 Basismodell: Keine Veränderung der Aufzeichnungszeitpunkte

Die Modellierung der Ereignisse zu den im Basismodell beschriebenen Zeitpunkten wird von den Wrappern übernommen. Während der Ausführungszeit hat ein Prozess nur seine lokalen Werte in seine Ereignisspur geschrieben. Lokale Bezeichner mit gleichem Wert können so auf verschiedenen Prozessen auch verschiedene Daten referenzieren. Es ist somit wichtig, für jedes Ereignis die lokalen Werte in global gültige Werte umzusetzen.

Eine EPILOG-Spur ist in zwei große Abschnitte geteilt. Im ersten Abschnitt werden alle Hilfsstrukturen zur Minimierung des Datenvolumens gespeichert. Diese Strukturen heißen *Definitionseinträge* (definition records). Sie dienen dazu Redundanzen bei der Speicherung der Ereignis-Daten zu vermeiden. Sie sind global in der Spur gültig und bilden selbst keine Ereignisse. Das bedeutet, dass sie weder einen Ort, noch einen Zeitstempel tragen. Während der Ausführung kann ein Prozess allerdings nicht vor Beginn der Aufzeichnung wissen, welche Datenstrukturen für den aufzuzeichnenden Programmablauf benötigt werden. Deshalb stehen Definitionseinträge und Ereignisse gemischt in der prozesslokalen Spur. Um an dieser Stelle Übersichtlichkeit zu schaffen, werden alle Definitionseinträge aus der lokalen Ereignisspur gesammelt und nach einem ersten Durchlauf durch alle Spuren gesammelt in die Ausgabespur geschrieben. Während dieses ersten Durchlaufs durch die prozesslokalen Spuren, werden die lokalen Bezeichner durch globale Bezeichner ersetzt und gleichzeitig

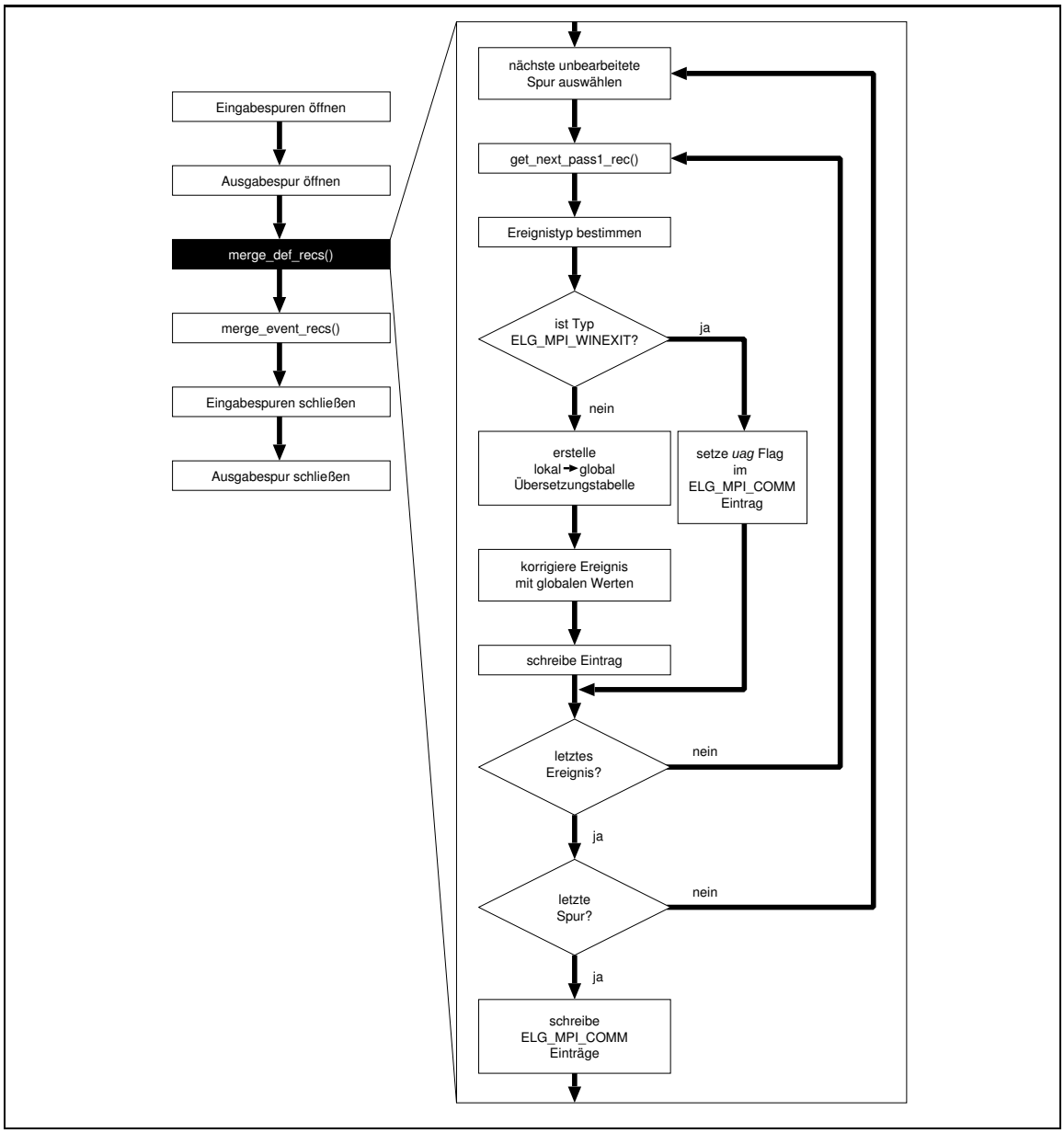


Abbildung 5.1: Schema des ersten Durchlaufs der Zusammenführung

Übersetzungstabellen erstellt, um bei der Umsetzung der späteren Ereigniseinträge die lokalen Werte ebenfalls auf die korrekten globalen Werte setzen zu können. Ein Schema der Vorgänge dieses ersten Durchlaufs ist in Abbildung 5.1 dargestellt. Auf der linken Seite der Abbildung ist zur Hilfestellung der Ablauf der Zusammenführung der Spuren insgesamt aufgeführt. Auf der rechten Seite ist dann der Ablauf während der ersten Phase der Zusammenführung abgebildet.

Nachdem jede Eingabespur geöffnet wurde, werden diese nacheinander abgearbeitet. Die Funktion `get_next_pass1_rec` liefert dabei den nächsten Eintrag der Eingabespur zurück, der im ersten Durchlauf bearbeitet werden muss. Das sind grundsätzlich alle Definitionseinträge innerhalb der Spur plus die `ELG_MPI_WINEXIT`-Einträge, da diese, bzw. die darin enthaltenen Informationen, für die korrekte Modifikation des *uag*-Flags in den `ELG_MPI_COMM`-Einträgen benötigt werden. Da die Definitionseinträge bereits in die Ausgabespur geschrieben werden, bevor der zweite Durchlauf startet, ist diese gesonderte Behandlung eines Ereigniseintrages während der Definitionseintrags-Phase nötig geworden.

Wenn ein geeigneter Eintrag zurückgeliefert wird, muss der Ereignistyp bestimmt werden,

um daran die nötigen Änderungen und weiteren Schritte festzulegen. Falls das gelesene Ereignis vom Typ `ELG_MPI_WINEXIT` ist, wird der in diesem Austritts-Ereignis referenzierte `ELG_MPI_COMM`-Eintrag angepasst. An dem Ereigniseintrag werden keine Änderungen vorgenommen.

Falls es sich um einen Definitionseintrag handelt, werden die lokalen Werte mit bereits gelesenen Werten verglichen und bei Bedarf für die lokalen Bezeichner globale Bezeichner eingeführt. Ein Kommunikator wird in jeder Einzelspur jedes Prozesses explizit definiert, muss aber in der Ausgabespur nur einmal gespeichert werden. Die lokalen Bezeichner können für den gleichen globalen Eintrag lokal zum Prozess unterschiedlich sein. Deshalb ist es nötig, für jede Einzelspur eine eigene Übersetzungstabelle zu führen, die für jeden Bezeichner die entsprechende Umsetzung aufnehmen kann. Nachdem die lokalen Werte in der Übersetzungstabelle vermerkt sind, wird der Definitionseintrag in die Ausgabespur geschrieben. Dies allerdings nur, wenn die darin enthaltenen Definitionen nicht redundant sind. Ein Kommunikator wird somit nur beim ersten Vorkommen in einer Spur verändert und wieder ausgegeben. Alle weiteren lokalen Definitionen dieses Kommunikators werden nicht mehr in der Ausgabespur gespeichert.

Wenn das Ende einer Spur erreicht ist, wird die nächste Spur verarbeitet. Wenn es keine weiteren Spuren mehr gibt, können die bis dahin zurückgehaltenen `ELG_MPI_COMM`-Einträge gesammelt geschrieben und dann die erste Phase abgeschlossen werden. Die Definitionseinträge werden mit den zwei letzten Einträgen `ELG_NUM_EVENTS`, welcher die Anzahl der in der Spur modellierten Ereignisse beinhaltet, und dem darauf folgenden `ELG_LAST_DEF`-Eintrag abgeschlossen.

Danach folgen in chronologischer Reihenfolge die Ereignisse der einzelnen Ereignisspuren. Der Ablauf während dieser zweiten Phase der Zusammenführung, bei der die Ereigniseinträge in die Ausgabespur geschrieben werden, ist in Abbildung 5.2 dargestellt.

Die Zeiger auf die Einträge der Einzelspuren werden vor dem zweiten Durchlauf wieder auf den Anfang der jeweiligen Spur gesetzt. Durch die chronologische Abfolge der Ereignisse innerhalb der Einzelströme, kann nun eine prozessübergreifende chronologische Reihenfolge aller Ereigniseinträge erstellt werden, in dem immer der Eintrag mit dem kleinsten Zeitstempel zur Bearbeitung aus einer der Einzelspuren zur Verarbeitung genommen wird. Der entsprechende Zeiger der Einzelspur wird auf das nächste Ereignis in seiner Spur gesetzt. Die Funktion `get_next_pass2_rec` liefert diesen Ereigniseintrag mit bereits umgewandelten Attributwerten. D.h. innerhalb der Funktion werden die lokalen Attributwerte anhand der im ersten Durchlauf erstellten Übersetzungstabellen auf die globalen Attributwerte gesetzt. Mit den globalen Werten kann der Ereigniseintrag weiterverarbeitet werden. Der Zeitstempel des Eintrags wird angepasst und relativ zum ersten Ereigniseintrag gesetzt. Der erste Ereigniseintrag erhält den Zeitstempel `timestamp = 0` als Referenzpunkt für den Beginn des Programmablaufs. Der frühere Attributwert wird in `start_time` gesichert. Alle weiteren Zeitstempel der Ereigniseinträge werden entsprechend modifiziert.

```
timestamp = timestamp - start_time;
```

Danach wird wieder der Ereignistyp bestimmt, um zu entscheiden, ob weitere Eingriffe in den Ereigniseintrag nötig sind. An dieser Stelle werden die entfernten Ereignisse in die entsprechenden lokalen Ereignisse umgesetzt, indem die Werte des Attributs `location` eines Eintrages mit dem Wert des Attributs `dlid` bzw. `slid` des Eintrages vertauscht werden. Zusätzlich wird der Typ des Ereignisses auf das entsprechende lokale Ereignis gesetzt. Abhängig vom Ereignistyp werden noch für eine eventuell anstehende Verschiebung Vorbereitungen getroffen.

An dieser Stelle wird zwischen dem Basismodell und dem erweiterten Modell unterschieden. Wenn durch die Umgebungsvariable `ELG_VT_MODE=1` angegeben ist, dass das Basismodell verwendet werden soll, werden an dieser Stelle alle weiteren Schritte übersprungen und

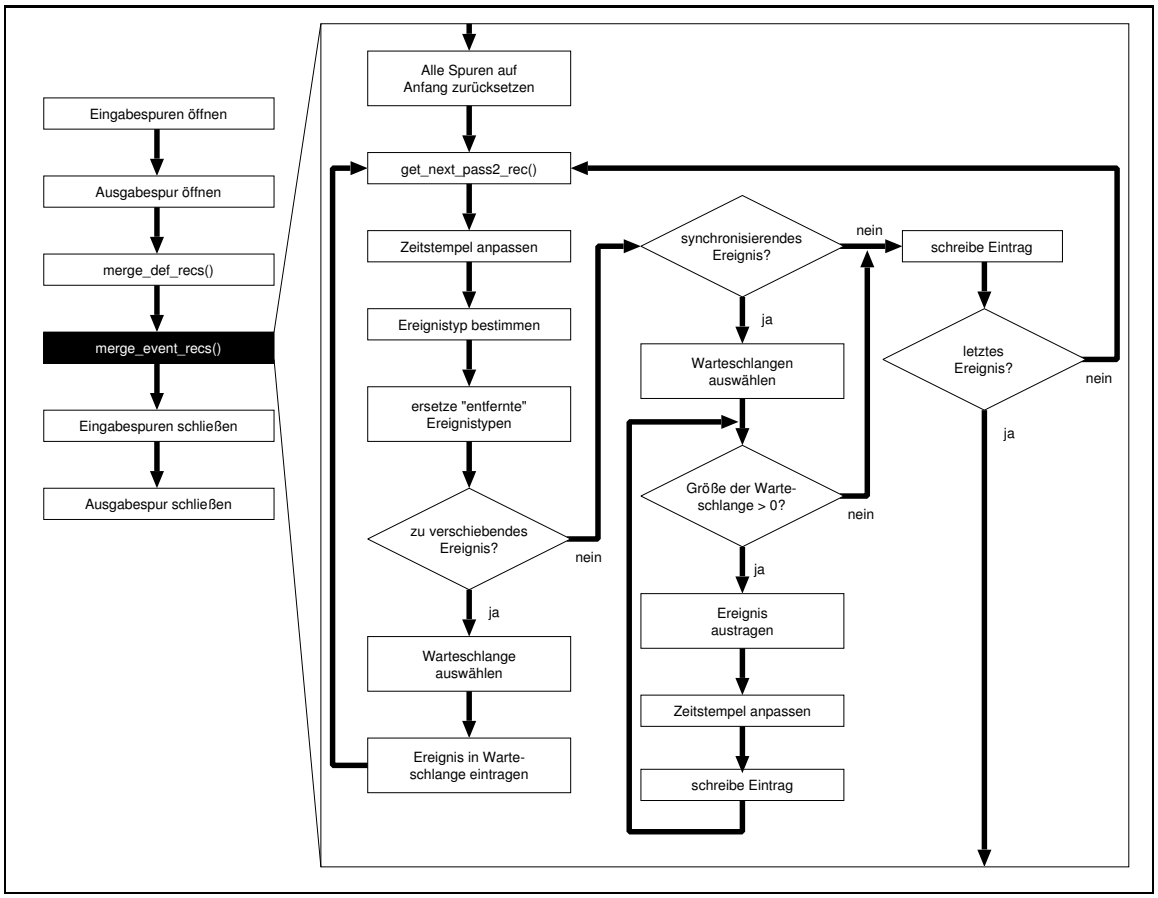


Abbildung 5.2: Schema des zweiten Durchlaufs der Zusammenführung

der Eintrag in die Ausgabespur geschrieben. Danach wird mit dem nächsten Eintrag fortgefahren. Durch `ELG_VT_MODE=0` oder keinerlei Angabe der Umgebungsvariable wird eine Umsetzung in das erweiterte Ereignismodell vorgenommen. Dazu wird wie folgt fortgefahren.

5.3.2 Erweitertes Modell: Verschiebung der Datentransferendpunkte

Ist der aktuelle Ereigniseintrag weder ein zu verschiebendes Ereignis, noch ein synchronisierendes Ereignis der einseitigen Kommunikation, dann wird der Eintrag ohne weitere Verzögerung geschrieben. Ist der Eintrag ein zu verschiebendes Ereignis, wird anhand der Attributwerte eine Warteschlange ausgewählt und der Eintrag angefügt. Danach wird mit dem nächsten Ereigniseintrag fortgefahren. Ist der Ereigniseintrag ein synchronisierendes Ereignis der einseitigen Kommunikation, dann wird anhand der Attributwerte eine oder mehrere Warteschlangen bestimmt und deren Größe abgefragt. Die Einträge werden nacheinander aus dieser Warteschlange entfernt und in die Ausgabespur geschrieben. Dabei wird der Zeitstempel auf die Zeit des synchronisierenden Ereigniseintrags gesetzt. Sind alle Einträge der betroffenen Warteschlangen in die Ausgabespur geschrieben, wird der aktuelle Ereigniseintrag des synchronisierenden Ereignisses in die Ausgabespur geschrieben, und mit dem nächsten Ereigniseintrag fortgefahren. Sind auf diese Weise alle Ereigniseinträge aller Einzelspuren verarbeitet worden, ist die zweite Phase der Zusammenführung abgeschlossen und alle beteiligten Spuren können geschlossen werden.

Die Verschiebung der Datentransferendpunkte vom Basismodell zum erweiterten Modell ist aufgrund der Tatsache, dass alle Verschiebungen in die Zukunft geschehen müssen, durch Warteschlangen zu lösen. Da das Programm `elg_merge` in C++ implementiert ist, können dafür die Konstrukte der STL (standard template library) verwendet werden. Sie stellt ver-

schiedene Funktionalitäten wie assoziative Felder und Warteschlangen durch das generische Programmierungsparadigma zur Verfügung.

Die Warteschlangen selbst werden durch den Datentyp `deque<ElgRec*>` realisiert, also eine doppelendige Warteschlange aus Zeigern auf `ElgRec`-Strukturen. Die Elemente der Warteschlange werden am Ende hinzugefügt und am Anfang entnommen. Dadurch wird die Reihenfolge der Ereignisse trotz gleicher Zeitstempel in der Ausgabespur erhalten. Jeder Prozess muss allerdings seine eigene Warteschlange führen. Gleichzeitig muss jeder Prozess in der Warteschlange zwischen den einzelnen existierenden Speicherfenstern unterscheiden. Diese werden durch Fenster-Bezeichner und Fenster-Lokation eindeutig bestimmt. Zusätzlich muss weiter unterschieden werden, ob es sich um das Ende einer Get- oder Put-Operation handelt. Die Ereignisse einer Get-Operation werden in der Datenstruktur `evt_get_q` gespeichert. Die Ereignisse einer Put-Operation werden in der Datenstruktur `evt_put_q` gespeichert. Daraus ergibt sich folgende Definition der Warteschlangen:

```
map<pair<elg_ui4,elg_ui4>,map<elg_ui4, deque<ElgRec*> > > evt_put_q;
map<pair<elg_ui4,elg_ui4>,map<elg_ui4, deque<ElgRec*> > > evt_get_q;
```

Die Auswahl der richtigen Warteschlange erfolgt durch die Angabe eines Tupels aus Speicherfenster-Bezeichner und Speicherfenster-Lokation, welches ein assoziatives Feld zurückliefert. In diesem Feld sind anhand der Prozess-Lokation die Warteschlangen gespeichert. Abbildung 5.3 zeigt eine schematische Darstellung der zusammengesetzten Datenstruktur.

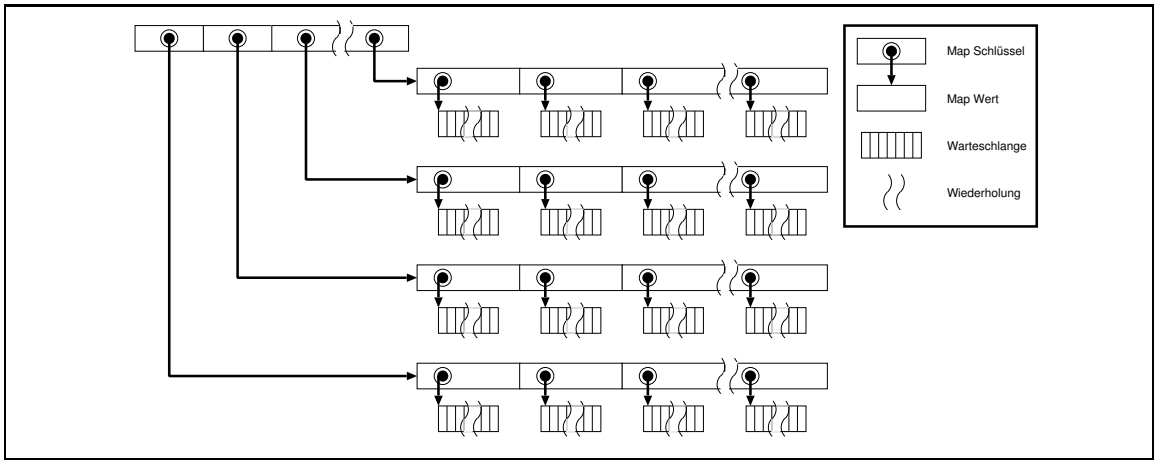


Abbildung 5.3: Schema der Warteschlangen-Datenstruktur des erweiterten Modells

Aus der Darstellung geht hervor, dass es sich insgesamt um ein zweidimensionales Feld handelt, dessen Elemente die Warteschlangen bilden. Der Zugriff auf dieses Feld kann in einfacher Weise durch den `[]`-Operator realisiert werden.

Wenn ein `ELG_MPI_WINEXIT`-Ereignis eines `MPI_Win_wait`-Aufrufs auf Prozess 0 bearbeitet wird, ist die entsprechende Warteschlange beispielsweise für das Fenster mit `wid = 1` durch folgenden Aufruf gegeben:

```
evt_get_q[make_pair(1,0)][0]
```

Die Fenster-Lokation und die Prozess-Lokation bei der Warteschlange `evt_get_q` ist immer identisch, da in dieser Warteschlange die Transfer-Ereignisse stehen, die an diesem Prozess ankommen. D.h. ein Prozess wird immer nur seine eigenen Warteschlangen bearbeiten. Um einen einheitlichen Zugriff auf die Warteschlangen zu geben, wird diese redundante Information im Schlüssel der assoziativen Felder weiterhin geführt. Während der Bearbeitung des `ELG_MPI_WINEXIT`-Ereignisses eines `MPI_Win_complete`-Aufrufs auf Prozess 1, welches sich auf das Fenster 5 des Prozesses 0 bezieht, würde folgende Warteschlange gewählt:

```
evt_get_q[make_pair(5,0)][1]
```

Ein Prozess überprüft immer die eigenen Warteschlangen, wenn für ihn ein lokales synchronisierendes Ereignis bearbeitet werden muss. Die einzige Ausnahme hiervon bildet eine Kombination aus getätigten Put-Operationen in Verbindung mit der Synchronisation mit passivem Ziel. Bei dieser Kombination ist der Abschluss des Datentransfers auf dem Zielprozess nicht an ein Ereignis auf dem Zielprozess, sondern an das `ELG_MPI_WIN_UNLOCK`-Ereignis auf dem Ursprungsprozess gekoppelt. Aus der Warteschlange des Zielprozesses dürfen aber nur diejenigen Ereignisse entnommen werden, die durch den Ursprungsprozess selbst in der Warteschlange gespeichert wurden. Für diesen Zweck wird eine Hilfswarteschlange gebildet, die während der Entnahme der Warteschlangeneinträge alle die Einträge aufnimmt, die noch nicht in die Ausgabespur geschrieben werden dürfen. Diese Warteschlange ersetzt die original Warteschlange nach deren vollständiger Bearbeitung und der Speicherung aller passenden Ereignisse in der Ereignisspur.

Im erweiterten Modell, sind die zu verschiebenden Ereignisse vom Typ `ELG_MPI_PUT_1TE` und `ELG_MPI_GET_1TE`, da das Ende des Datentransfers den spätesten Abschluss der Operation markiert, und dieser innerhalb der Synchronisation liegen muss. Ist das aktuell zu bearbeitende Ereignis von einem dieser Typen wird es bis zum nächsten zugehörigen synchronisierenden Ereignis der einseitigen Kommunikation in der entsprechenden Warteschlange zwischengespeichert.

Zu den synchronisierenden Ereignissen zählen das `ELG_MPI_WINCOLLEXIT`-Ereignis der `MPI_Win_fence`-Aufrufe, das `ELG_MPI_WINEXIT`-Ereignis mit gesetztem `synex`-Flag und das `ELG_MPI_UNLOCK`-Ereignis.

5.4 Änderungen für die Analysephase

Mit den in der Messphase gesammelten Daten im Ereignisstrom, muss für die einseitige Kommunikation nun die Schnittstelle erweitert werden, die der Analyseumgebung EXPERT den Zugriff auf die einzelnen Ereignisse innerhalb des Ereignisstroms ermöglicht. Zur Erweiterung dieser Schnittstelle ist eine Anpassung der Klassenhierarchie in EARL notwendig, um die neuen Ereignisklassen in das bestehende Konzept einzugliedern. Zusätzlich werden für die Bereitstellung der benötigten Informationen für die erweiterten Leistungsmerkmale Hilfsstrukturen benötigt. Eine vollständige Implementierung würde den Rahmen dieser Diplomarbeit sprengen. Deshalb werden die notwendigen Arbeiten hier nur skizziert. Abbildung 5.4 zeigt die neue Hierarchie der Ereignistypen, anhand dessen die Ereignistypen der einseitigen Kommunikation in die bestehende Klassenhierarchie integriert werden müssen.

Event beschreibt den grundlegenden Ereignistyp. Die weiteren Ereignistypen bilden eine Spezialisierung dieses Basistyps und ermöglichen eine Zusammenfassung der einzelnen Ereignistypen zu kontextbezogenen Gruppen. Von diesen Gruppen werden speziellere Ereignistypen zur Kontrollflussmodellierung (FLOW), Nachrichtenversendung (P2P), Threaderstellung (TEAM) und Threadsynchronisation (SYNC) abgeleitet.

Ein Ereignisstrom definiert sich als endliche indizierte Menge von Ereignissen. Die Indizierung wird durch chronologische Sortierung ermöglicht. Gleichzeitig wird definiert, dass innerhalb eines Ereignisstroms, Ereignisse des gleichen Prozesses oder Threads unterschiedliche Zeitpunkte haben müssen. Das bedeutet, Ereignisse können innerhalb des gleichen Prozesses oder Thread nicht gleichzeitig geschehen.

Unterhalb des Basisereignistypen Event wurde eine neue Gruppe RMA geschaffen. Hier werden bis auf spezielle EXIT-Ereignistypen, die zu den Kontrollflussereignistypen zählen, alle RMA-spezifischen Ereignistypen zusammengefasst. Um sowohl Datenfluss als auch Initiative des Datentransports modellieren zu können, werden für die RMA-Operationen fünf Ereignistypen definiert. `MPI_Put` und `MPI_Accumulate` werden durch PUT_1TS und

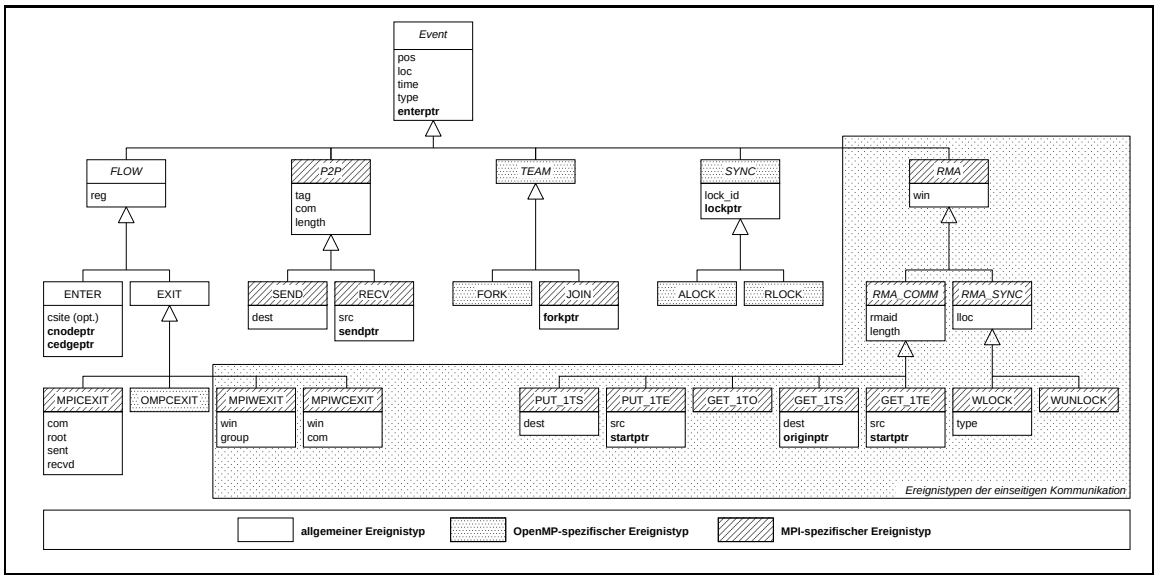


Abbildung 5.4: Erweiterte Struktur der Ereignistypen in EARL 2.1

PUT_1TE, `MPI_Get` durch GET_1TO, GET_1TS und GET_1TE modelliert. Entfernte Ereignistypen werden nicht mehr benötigt, da in dieser Phase bereits eine globale Sicht auf die Ereignisse existiert. WLOCK und WUNLOCK modellieren die bei der Synchronisation mit passivem Ziel nötigen Sperren. Insbesondere um die Synchronisation innerhalb der einseitigen Kommunikation zu verfolgen, ist es ebenfalls nötig, zwei neue EXIT-Ereignistypen zu definieren. In Anlehnung zu dem Ereignistyp EXIT bestimmt MPIWEXIT das Verlassen einer Region, die einen Bezug zur einseitigen Kommunikation hat. MPIWCEXIT übernimmt die gleiche Funktion für kollektive Aufrufe innerhalb der einseitigen Kommunikation, also Erschaffung und Freigabe von Speicherfenstern sowie die Synchronisation mittels Fence.

Auf der Ebene der Analyse sind die Prozessgruppen in den Hintergrund gerückt. Die Informationen, die aus den aufgezeichneten Daten herausgearbeitet werden können, dienen der Zuordnung der beteiligten Prozesse an einer Synchronisation oder RMA-Operation selbst.

Die Implementation der Ereignisklassen in EARL erfolgt in zwei Stufen. Die Attribute und Methoden aller Ereignisklassen werden in der Klasse `earl::Event` definiert und implementiert. Die eigentlichen Ereignisklassen bilden eine Repräsentation dieser Ereignisklasse. Für die bestehenden Ereignisklassen ergeben sich daher folgende Repräsentationen:

- `class Event_rep`
- `class Flow_rep : public Event_rep`
- `class Enter_rep : public Flow_rep`
- `class Exit_rep : public Flow_rep`
- `class MPICExit_rep : public Exit_rep`
- `class OMPCExit_rep : public Exit_rep`
- `class P2P_rep : public Event_rep`
- `class Send_rep : public P2P_rep`
- `class Recv_rep : public P2P_rep`
- `class Team_rep : public Event_rep`
- `class Fork_rep : public Team_rep`
- `class Join_rep : public Team_rep`
- `class Sync_rep : public Event_rep`
- `class Alock_rep : public Sync_rep`
- `class Rlock_rep : public Sync_rep`

Für die einseitige Kommunikation müssen somit folgende zusätzliche Repräsentationen für die einzelnen Ereignisklassen geschaffen werden:

- `class MPIWCExit_rep : public Exit_rep`
- `class MPIWExit_rep : public Exit_rep`
- `class RMA_rep : public Event_rep`
- `class RMA_Comm_rep : public RMA_rep`
- `class Put_1TS_rep : public RMA_Comm_rep`
- `class Put_1TE_rep : public RMA_Comm_rep`
- `class Get_1TS_rep : public RMA_Comm_rep`
- `class Get_1TE_rep : public RMA_Comm_rep`
- `class Get_1TO_rep : public RMA_Comm_rep`
- `class RMA_Sync_rep : public RMA_rep`
- `class Wlock_rep : public RMA_Sync_rep`
- `class Wunlock_rep : public RMA_Sync_rep`

In Beispiel 5.2 wird die Ereignisklasse `MPIWCExit_rep` exemplarisch für die restlichen Ereignisklassen dargestellt. Der Aufbau der anderen Klassen wäre analog.

Neben dem Konstruktor, der zusätzlich einen Parameter für den Speicherfenster-Bezeichner und den benutzten Kommunikator enthält, sind noch die Zugriffsfunktionen auf diese Attribute implementiert. Die Attribute selbst sind als `private` deklariert. Um redundante Informationen zu vermeiden, werden sind diese Attribute nur Zeiger auf die entsprechenden Objekte, da sowohl `Window` als auch `Communicator` eine eigene Klasse bilden, um komplexere Zusammenhänge abbilden zu können.

Durch die Implementation der Austrittsereignisse `MPIWCExit_rep` und `MPIWExit_rep` wird bereits die Ermittlung der einfachen Leistungsmerkmale der einseitigen Kommunikation sowie die später definierten ersten musterbasierten Leistungsprobleme für die einseitige Kommunikation in der Analysephase ermöglicht. Die weiteren Ereignisklassen unterhalb von `RMA_rep` sind nötig, um die vollen Möglichkeiten der Modellierung weiterer musterbasierten Leistungsprobleme in der Zukunft zu unterstützen. Musterbasierte Leistungsprobleme werden im Kapitel 6 beschrieben.

Neben dem wahlfreien Zugriff auf Ereignisse und Attribute, berechnet EARL auch Zustände und Zeigerattribute und stellt diese dem Anwender zur Verfügung. Deswegen müssen auch diese Teile von EARL für die einseitigen Kommunikation erweitert werden.

Um die neuen Zeigerattribute *startptr* und *originptr* berechnen zu können, werden neuen Zustände benötigt. Eine Transfer-Warteschlange für alle offenen einseitigen Datentransfers beinhalten. Das Verhalten dieser Warteschlange ist analog zur bereits bestehenden Warteschlange für Nachrichten der Punkt-zu-Punkt-Kommunikation. Ein Transfer-Start-Ereignis wird in die Warteschlange eingereiht, bis das korrespondierende Transfer-End-Ereignis gelesen wurde, und das Zeigerattribut gesetzt werden kann. Dann wird das entsprechende Transfer-Start-Ereignis aus der Warteschlange entfernt. Für Get-Operationen ist noch eine zusätzliche Warteschlange nötig. Die Ursprungs-Warteschlange nimmt alle Transfer-Origin-Ereignisse auf, bis der korrespondierende Transfer-Beginn bestimmt werden kann. Dann wird das *originptr*-Attribut des GET_1TS-Ereignisses gesetzt, das Ursprungsereignis aus der Warteschlange gelöscht und das GET_1TS-Ereignis kann in die Transfer-Warteschlange übergeben werden.

Eine umfassende Verfolgung der Zugriffs- und Freigabeepochen ist nun ebenfalls möglich. Dabei muss beachtet werden, dass innerhalb eines Programms verschiedene Synchronisationsmechanismen genutzt werden können. Dies kann zur Unterscheidung in RMA-Epochen

und lokale Epochen genutzt werden. Direkt nach einem `MPI_Win_fence`-Aufruf ist unklar, ob eine neue RMA-Epoche begonnen hat, bis die nächste RMA-Operation oder RMA-Synchronisation getätigt wurde. Dann ist nachträglich entscheidbar, um welche Art der Epoche es sich gehandelt hat. Dazu ist eine ähnliche Modellierung wie das Regionen-Modell denkbar, da sich lokale und RMA-Epochen wechselseitig ausschließen und zusammen aber die Gesamtheit der möglichen Epochenarten repräsentieren.

```
#include <string>
#include <vector>

#include "Exit_rep.h"

namespace earl
{
  class MPIWCExit_rep : public Exit_rep
  {
  public:

    MPIWCExit_rep(State& state,
 EventBuffer*  buffer,
 Location*     loc,
 double        time,
 std::map<long, Metric*>&  metm,
 std::vector<elg_ui8>& metv,
 Window*       win,
 Communicator* com) :
      Exit_rep(state, buffer, loc, time, metm, metv),
      win(win),
      com(com) {}

    virtual bool        is_type(etype type) const;
    virtual etype       get_type()    const { return MPIWCEXIT; }
    virtual std::string get_typestr() const { return "MPIWCEXIT"; }

    Window*        get_win()      const { return win; }
    Communicator*  get_com()      const { return com; }

    virtual void trans(State& state);

  private:

    Window*       win;
    Communicator* com;
  };
}
```

Beispiel 5.2: Die Ereignisklasse `MPIWCExit_rep`

Kapitel 6

Leistungsmerkmale der einseitigen Kommunikation

6.1 Einleitung

Durch die in Kapitel 4 entwickelten Ereignismodelle und die in Kapitel 5 beschriebenen Änderungen an der Infrastruktur zur Aufzeichnung und Analyse der Ereignisse der einseitigen Kommunikation, sind nun alle Grundlagen geschaffen, um darauf aufbauend Leistungsmerkmale definieren zu können. Die bisher in KOJAK definierten Leistungsmerkmale wurden in Abbildung 2.5 auf Seite 16 bereits dargestellt. Für die einseitige Kommunikation wird diese Hierarchie an Leistungsmerkmalen nun erweitert. Der resultierende Baum an Leistungsmerkmalen ist in Abbildung 6.1 dargestellt.

In der folgenden Beschreibung der Leistungsmerkmale wird eine Notation verwendet, die eine Einordnung in der in Abbildung 6.1 dargestellten Baumstruktur erleichtern soll. In der Abbildung beziehen sich Kindelemente mit einem Pfeil auf die Eltern. Dies wird bei der Beschreibung der Leistungsmerkmale durch das Symbol „←“ ausgedrückt. Der eigentliche Name des Leistungsmerkmals ist somit der letzte durch das ←-Symbol getrennte Bereich des Namens. Zur besseren Übersichtlichkeit innerhalb des fließenden Textes, werden Leistungsmerkmale nur soweit beschrieben, dass ihre Identifikation eindeutig gewährleistet ist. Synchronization alleine bietet beispielsweise keine eindeutige Identifizierung, da es einen Teilbaum mit diesem Namen sowohl unter MPI als auch unter OpenMP gibt. Durch MPI←Synchronization wird jedoch der Teilbaum der Hierarchie eindeutig bestimmt.

Die Leistungsmerkmale mit weißem Hintergrund sind *einfache Leistungsmerkmale* (profiling patterns). Sie zeichnen sich dadurch aus, dass in ihnen nur „einfach“ Metriken für eine bestimmte Region akkumuliert und diese dann zu Gruppen zusammengefasst werden. Zum Beispiel ist in MPI←Communication die gesamte Zeit akkumuliert, die der Kontrollfluss in Regionen der MPI-Funktionen zur Kommunikation liegt. Es ergibt sich hierbei eine Hierarchie, in der Leistungsmerkmale von anderen Leistungsmerkmalen beinhaltet werden können. Leistungsmerkmale mit grauem Hintergrund sind *musterbasierte Leistungsmerkmale* (complex patterns). Diese Leistungsmerkmale ergeben sich aus Beziehungen, die Ereignisse des Ereignisstroms untereinander haben. Mit Hilfe der Beziehungen zwischen den Ereignissen lassen sich Muster definieren, die auf ineffiziente Programmstrukturen hinweisen. Wenn beispielsweise im Ereignisstrom ein Enter-Ereignis einer `MPI_Recv`-Region vor dem Enter-Ereignis der zugehörigen `MPI_Send`-Region vorkommt, spricht man von einem Late Sender-Leistungsproblem. Der empfangende Prozess verbringt unnötig Rechenzeit mit dem Warten auf den Sendeprozess, da keine Daten empfangen werden können, bevor sie nicht beim sendenden Prozess losgeschickt wurden. Die bisher in EXPERT definierten musterbasierten Leis-

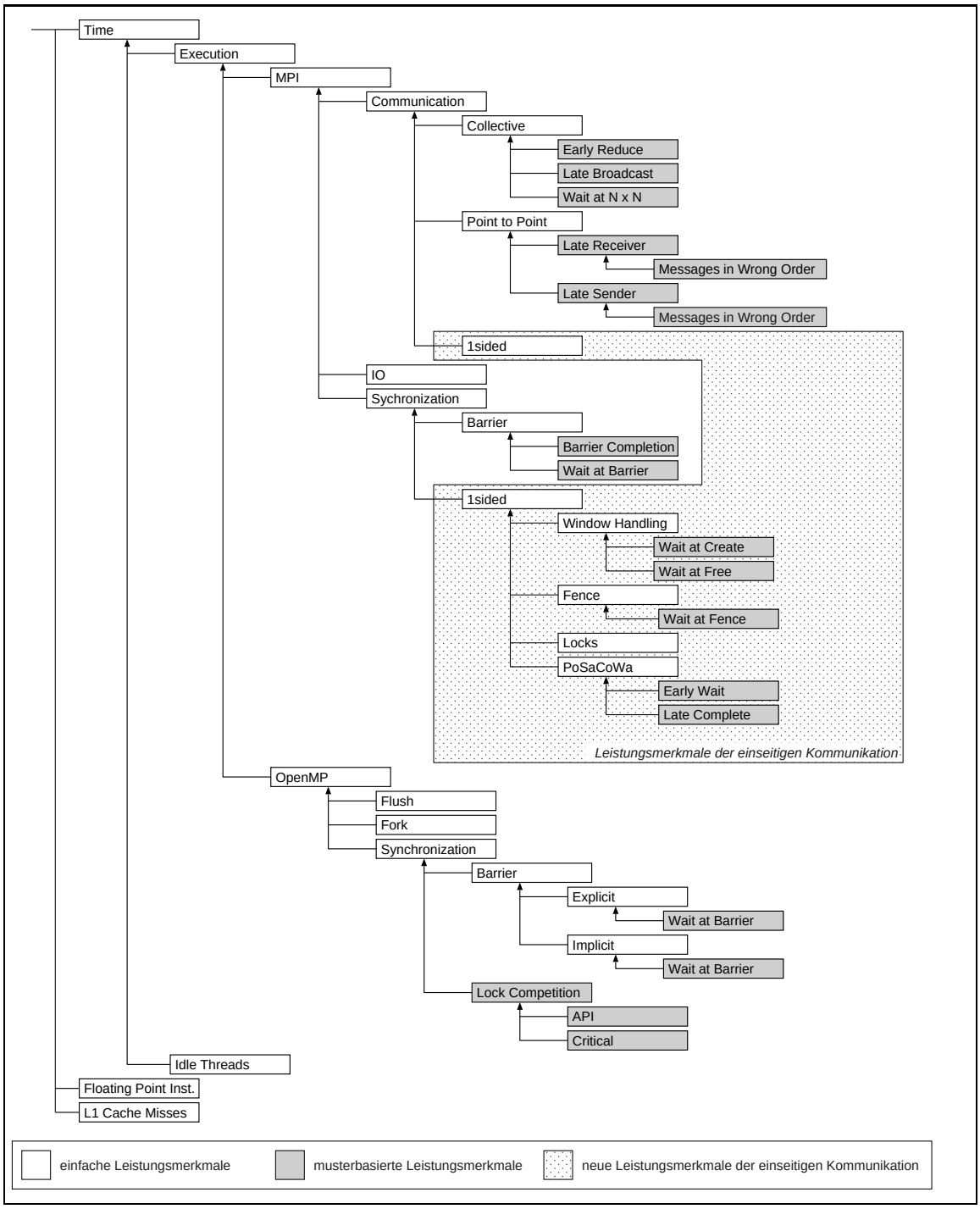


Abbildung 6.1: Hierarchie der Leistungsmerkmale in EXPERT 3.1

tungsmerkmale sind Leistungsprobleme, d.h. durch diese Muster innerhalb des Programmablaufs wird immer ein Zeitverlust ausgedrückt. Die meisten einfachen Leistungsmerkmale in EXPERT sind hingegen bewertungsfrei. Diese Zuordnung, dass musterbasierte Leistungsmerkmale ausschließlich Leistungsprobleme beschreiben und einfache Leistungsmerkmale bewertungsfrei sind, ist allerdings nicht zwingend, wie die Merkmale OpenMP←Flush und Idle Threads zeigen. Beide einfachen Leistungsmerkmale beschreiben ein Leistungsproblem. Bewertungsfrei bedeutet in diesem Zusammenhang, dass das Analyseprogramm nicht selbst entscheiden kann, ob es sich um ein Leistungsproblem handelt oder nicht. Es kann lediglich das Ausmaß (severity) dieses Leistungsmerkmals bestimmen. Es liegt am Anwender, die dort zusammengefassten Werte zu interpretieren und mit einem erwarteten Verhalten in

Zusammenhang zu bringen. Z.B. ist es für eine kommunikationsintensive Anwendung unter Umständen hervorragend nur 10 Prozent der gesamten Ausführungszeit mit Datentransfer zu verbringen. Für eine Anwendung mit sehr geringer Kommunikationslast kann dies allerdings schlecht sein.

Durch die Erweiterung der Leistungsmerkmale für die einseitige Kommunikation ergeben sich innerhalb des Merkmalbaums an verschiedenen Stellen Änderungen. Die einseitige Kommunikation beschreibt mit ihren RMA-Operationen Methoden zum Datentransfer. Ausführungszeiten der Funktionen zum expliziten Nachrichtenaustausch werden bisher in MPI←Communication zusammengefasst. Die RMA-Operationen der einseitigen Kommunikation bilden deshalb hier einen neuen Unterpunkt.

Da die Semantik der einseitigen Kommunikation die Synchronisation vom Datentransfer trennt, sind dafür auch getrennte einfache Leistungsmerkmale vorgesehen. Synchronisation innerhalb von MPI besitzt bereits ein definiertes Leistungsmerkmal. Deshalb wird, die Synchronisation der einseitigen Kommunikation ebenfalls dort beschrieben. Die enge Verflechtung der Synchronisation mit dem Datentransfer drückt sich in der neuen Hierarchie dadurch aus, dass alle späteren musterbasierten Leistungsmerkmale der einseitigen Kommunikation innerhalb des Zweiges von MPI←Synchronization←1sided definiert sind. Dies ist mit dadurch bedingt, dass die Ende-Ereignisse der Datentransfers in den Regionen der Synchronisation liegen und die Datentransferfunktionen selbst meist nicht-blockierend sind.

Um den unterschiedlichen Synchronisationsmechanismen der einseitigen Kommunikation in MPI auch bei den Leistungsmerkmalen Rechnung tragen zu können, werden unterhalb von MPI←Synchronization weitere einfache Leistungsmerkmale definiert, die jeweils eines der Synchronisationsmethoden repräsentieren. Zusätzlich kommt dazu noch ein einfaches Leistungsmerkmal, welches Zeiten der Speicherfensterverwaltung von MPI zusammenfasst. Diese Funktionen dienen zwar nicht der expliziten Daten- oder Zugriffssynchronisation der Prozesse, doch durch eine implizite Barrier-Synchronisation innerhalb dieser Funktionen gewähren sie zumindest die zeitliche Synchronisation der Prozesse. Der Begriff Synchronisation bezieht sich hierbei sowohl auf eine prozessübergreifende zeitliche Synchronisation der Ausführungszeiten bestimmter Gruppen von Funktionen, einer Zugriffssynchronisation der Prozesse auf gemeinsame Daten, als auch dem wechselseitigen Ausschluss und einer Datensynchronisation, bei dem beispielsweise verschiedene Instanzen einer Variable abgeglichen werden, um ein konsistentes Datenbild zu ergeben. Synchronisationsmechanismen, die eine dieser Arten der Synchronisation innerhalb der einseitigen Kommunikation gewährleisten, werden an dieser Stelle der Hierarchie beschrieben. Jede dieser einfachen Leistungsmerkmale beschreibt wiederum weitere musterbasierte Leistungsmerkmale, die speziell in ihrem Bereich auftreten können.

6.2 Einfache Leistungsmerkmale

Aus den zur Laufzeit gesammelten Daten lassen sich bei der Analyse des Ereignisstroms Daten über Ereignisse sinnvoll zusammenfassen. Diese Zusammenfassung geschieht im allgemeinen bezogen auf eine Gruppe von Regionen. So kann über diese Gruppe von Regionen aus den Spurdaten ein Laufzeitprofil erstellt werden. Alle einfachen Leistungsmerkmale, die für die einseitige Kommunikation im Rahmen dieser Arbeit definiert wurden, sind bewertungsfrei.

6.2.1 MPI Communication

Time←Execution←MPI←Communication←1sided

Das Leistungsmerkmal Communication←1sided enthält eine Akkumulation der Ausführungszeit der Regionen der RMA-Operationen `MPI_Put`, `MPI_Get` und `MPI_Accumulate`.

6.2.2 MPI Synchronization

Time←Execution←MPI←Synchronization

Hier werden die beiden einfachen Leistungsmerkmale MPI←Synchronization←Barrier für die kollektive Kommunikation und MPI←Synchronization←1sided für die einseitige Kommunikation zusammengefasst.

Wichtig Dieses Leistungsmerkmal ist im Zuge der Erweiterung um die einseitige Kommunikation nicht neu hinzugefügt worden, sondern wurde in seiner ursprünglichen Funktion verändert. Dies war notwendig, um die Synchronisation der kollektiven Kommunikation neben der Synchronisation der einseitigen Kommunikation hierarchisch korrekt einordnen zu können.

Time←Execution←MPI←Synchronization←Barrier

Die bisher in Synchronization zusammengefasste Ausführungszeit wird nun durch Barrier ausgedrückt. Das ist die akkumulierte Ausführungszeit aller Barrier-Regionen.

Time←Execution←MPI←Synchronization←1sided

Die Synchronisation der einseitigen Kommunikation ist eng in das Kommunikationskonzept eingeflochten. Synchronization←1sided bietet deshalb ein Leistungsmerkmal, in dem alle Ausführungszeiten der Synchronisation mit Fence, der allgemeinen Synchronisation mit aktivem Ziel und der Synchronisation mit passivem Ziel zusammengefasst werden. Zusätzlich werden auch die Ausführungszeiten, die für die Fenstererstellung nötig sind, mit einbezogen.

Time←Execution←MPI←Synchronization←1sided←Window Handling

Die Erstellung eines gemeinsamen Speicherfenster-Handles ist innerhalb von MPI eine zeitaufwendige Operation. Die Ausführungszeiten sollten möglichst minimiert werden. Zusätzliche und vor allem unnötige Ausführungszeit können großen Einfluss auf die Leistungsfähigkeit der Anwendung haben. Es ist meist nicht sinnvoll, innerhalb des Programmablaufs, häufig Fenster neu zu definieren und sie nach Gebrauch wieder freizugeben. Der Overhead für diese kollektiven Operationen wäre unnötig.

Die Ausführungszeiten der Funktionen `MPI_Win_create` und `MPI_Win_free` bilden dieses einfache Leistungsmerkmal. Weitere explizite Funktionen zur Verwaltung der Fenster sind von MPI nicht gegeben und auch nicht nötig. Die restlichen leistungsrelevanten Funktionen der Fensterverwaltung sind in den anderen Leistungsmerkmalen beschrieben.

Time←Execution←MPI←Synchronization←1sided←Fence

Die Synchronisation mit Fence ist eine der drei Synchronisationsmechanismen, die von MPI angeboten werden. Die Synchronisation mit Fence ist ein kollektiver Aufruf über mehrere Prozesse. In diesem Leistungsmerkmal wird die kollektive Ausführungszeit innerhalb der Aufrufe von `MPI_Win_fence` festgehalten.

Time←Execution←MPI←Synchronization←1sided←Locks

Die Synchronisation mit passivem Ziel benutzt Sperren. Durch sie werden die Freigabe- und Zugriffsepochen auf die Speicherfenster der Prozesse koordiniert. Die Ausführungszeiten der Funktionen `MPI_Win_lock` und `MPI_Win_unlock` werden in diesem Leistungsmerkmal beschrieben.

Time←Execution←MPI←Synchronization←1sided←PoSaCoWa

Der Name dieses Leistungsmerkmals beruht auf den Funktionen der allgemeinen Synchronisation mit aktivem Ziel. Dies sind: `MPI_Win_post`, `MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_wait` sowie `MPI_Win_test`.

6.3 Musterbasierte Leistungsmerkmale

Musterbasierte Leistungsmerkmale können prozesslokale aber auch prozessübergreifende Ereignismuster sein, die den Teil eines einfachen Leistungsmerkmals abbilden, der durch ineffiziente Kommunikation oder Berechnungen verursacht wurde. Alle für die einseitige Kommunikation definierten musterbasierten Leistungsmerkmale sind gleichzeitig Leistungsprobleme. Diese sollten idealer Weise minimiert werden. Der ideale dynamische Ablauf eines Programms enthält nur Leistungsmerkmale und keine Leistungsprobleme.

Time←Execution←MPI←Synchronization←1sided←Window Handling←Wait at Create

In diesem Leistungsproblem sind alle Prozesszeiten akkumuliert, die die am kollektiven Erzeugen einer Fensterreferenz durch `MPI_Win_create` beteiligten Prozesse in den Aufruf eintreten, bevor der letzte Prozess eintritt. Es ist analog zum Leistungsproblem Wait at Fence, welches in Abbildung 6.2 dargestellt ist.

Time←Execution←MPI←Synchronization←1sided←Window Handling←Wait at Free

In diesem Leistungsproblem sind alle Prozesszeiten akkumuliert, die die an der kollektiven Freigabe einer Fensterreferenz durch `MPI_Win_free` beteiligten Prozesse in den Aufruf eintreten, bevor der letzte Prozess eintritt. Es ist Es ist analog zum Leistungsproblem Wait at Fence, welches in Abbildung 6.2 dargestellt ist.

Time←Execution←MPI←Synchronization←1sided←Fence←Wait at Fence

Die Synchronisation mit Fence wurde in Kapitel 3 vorgestellt als Synchronisationskonzept für Anwendungen, die in ihrem Programmablauf alternierende gemeinsame Kommunikations- und Berechnungsphasen haben. Diese Phasen müssen nicht bei allen Prozessen der parallelen Anwendung zeitlich übereinstimmen, allerdings für die Prozesse, die während der Kommunikationsphase auf ein gemeinsam definiertes Speicherfenster zugreifen. Wenn die Prozesse der Anwendungen zum Beispiel durch eine ungenügende Lastverteilung der Berechnungsphase zu unterschiedlichen Zeitpunkten in die Synchronisation eintreten, müssen diese Prozesse unnötig auf die restlichen Prozesse warten. Diese Wartezeit ist im Leistungsproblem Wait at Fence zusammengefasst, welches analog zu dem in den bisherigen Leistungsproblemen bekannten Wait at NxN entworfen ist. Eine Darstellung ist in der folgenden Abbildung 6.2 gegeben.

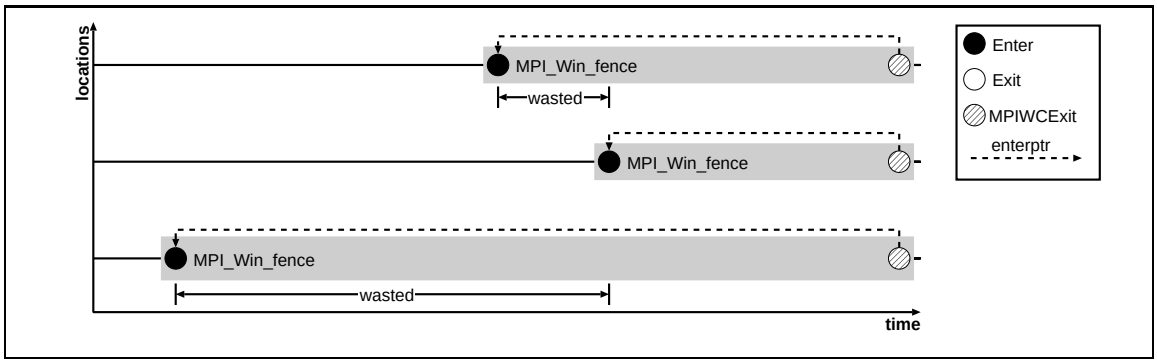


Abbildung 6.2: Das Leistungsproblem Wait at Fence

Time←Execution←MPI←Synchronization←1sided←PoSaCoWa←Late All Complete

Da die Funktion `MPI_Win_wait` auf dem Zielprozess bis zum Abschluss der getätigten RMA-Operation blockiert, ist die verfügbare Rechenzeit unnötig gebunden, wenn die Ursprungsprozesse der RMA-Operationen noch nicht den eigenen Abschluss mittels `MPI_Win_complete` eingeleitet haben. Eine graphische Beschreibung dieses Leistungsproblems ist in der folgenden Abbildung 6.3 angegeben. Dieses Leistungsproblem kann zum einen von den Ursprungsprozessen verursacht sein, wenn dort das Late Complete-Leistungsproblem auftritt, zum anderen aber auch durch ungünstige Lastverteilung des Benutzercodes, wenn dadurch der Zielprozess den Aufruf von `MPI_Win_wait` zu früh startet. In diesem Fall kann man auch von einem Early Wait sprechen.

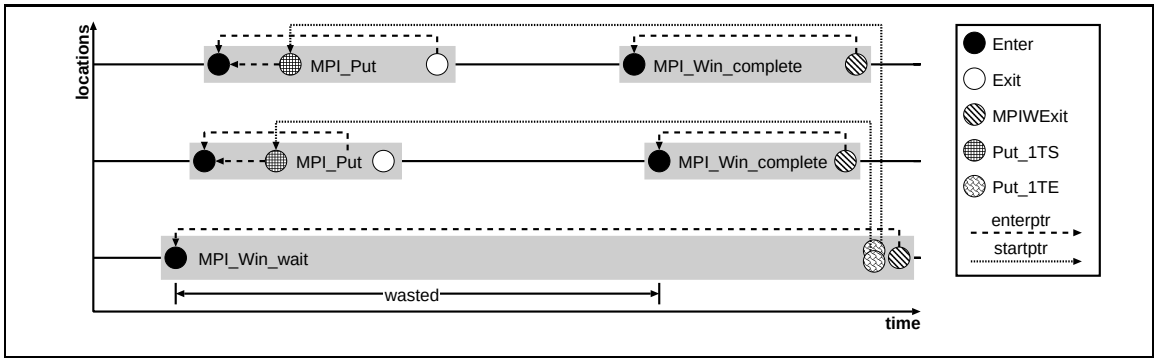


Abbildung 6.3: Das Leistungsproblem Late All Complete bzw. Early Wait

Time←Execution←MPI←Synchronization←1sided←PoSaCoWa←Late Complete

Late Complete gibt in der Verbindung mit Idle Wait an, wieviel Zeit unnötiger Weise gebraucht wurde, weil die letzte RMA-Operation auf dem Fenster nicht unmittelbar durch `MPI_Win_complete` abgeschlossen wurde. Im Allgemeinen sollten die abschließenden Syn-

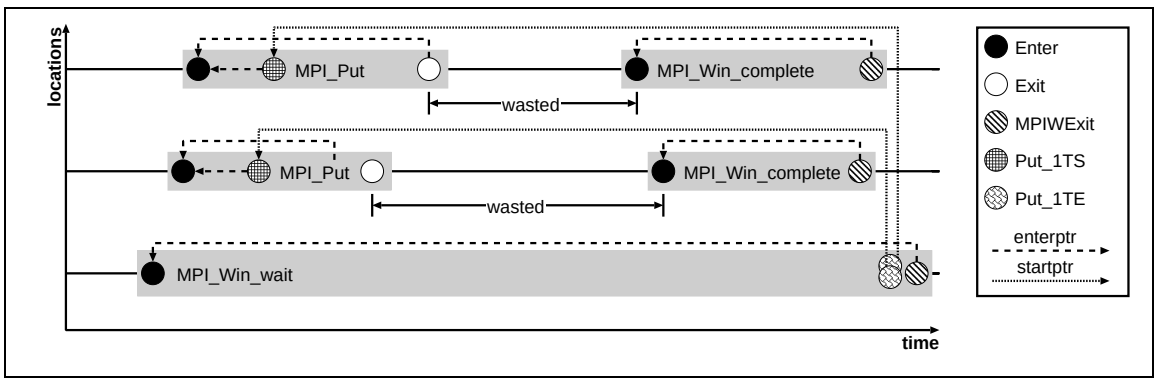


Abbildung 6.4: Das Leistungsproblem Late Complete

chronisationsfunktionen so eng wie möglich um die zugehörigen RMA-Operationen liegen, um diese Zeiten zu vermeiden. Aufgrund dieser Definition ist für den Anwender die Ursache für ineffizientes Programmverhalten beim Ursprungsprozess zu suchen. Eine Darstellung der Zusammenhänge des Leistungsproblems ist in der folgenden Abbildung 6.4 angegeben.

Natürlich sind noch weitere (komplexere) Merkmale für die einseitige Kommunikation denkbar. Die hier beschriebenen Merkmale bilden jedoch eine gute Grundlage, um Erfahrungen für die Analyse der einseitigen Kommunikation zu sammeln und daraus weitere Merkmale zu entwickeln.

Kapitel 7

Visualisierung der einseitigen Kommunikation

7.1 Motivation

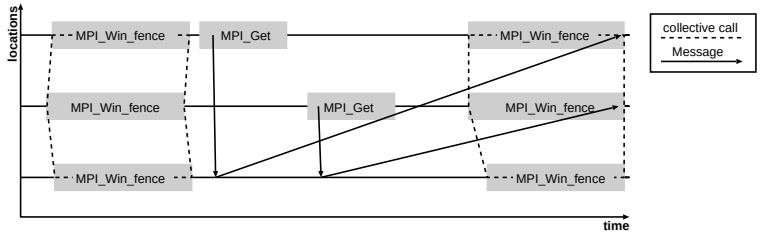
Die Visualisierung der Leistungsdaten ist ein Grundbaustein für den Anwender, einen schnellen Überblick über das Laufzeitverhalten seines Programms zu erlangen. Im Bereich der ereignisbasierten Leistungsanalyse durch Tracing gehört dazu eine grafische Darstellung der Ereignisse in einem Zeitlinien-Diagramm. Daran kann der Anwender schnell einen Teilbereich des Programmablaufs überblicken. Dabei ist es wichtig, logische Zusammenhänge zwischen den Ereignissen hervorzuheben, sodass sie vom Anwender leicht optisch aufzunehmen sind.

Neben der weiterhin möglichst genauen Abbildung der realen Ereignisse im Programmablauf durch modellierte Ereignisse, ist es wichtig implizite Zusammenhänge des Ereignisstroms dem Anwender explizit darzustellen. Das bedeutet, dass besonders Zusammengehörigkeiten und Abhängigkeiten von verschiedenen Ereignissen dem Anwender verdeutlicht werden müssen.

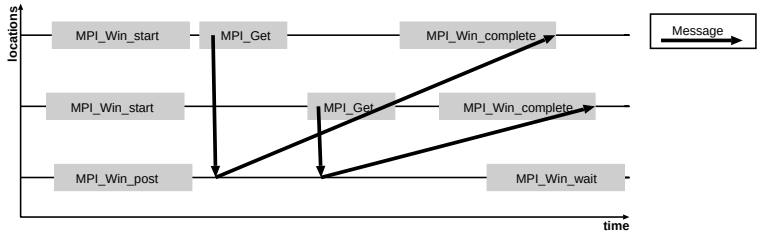
Die im Laufe des Kapitel vorgestellten Ansätze zur Visualisierung der einseitigen Kommunikation und ihre Konsequenzen für das darunterliegende Ereignismodell wurden in starkem Maße durch Gespräche mit Rolf Rabenseifner [14] inspiriert und beeinflusst. Da KOJAK als Analyseumgebung keine eigenständige Visualisierungskomponente für Ereignisströme besitzt, ist es auf die Fähigkeiten anderer Werkzeuge angewiesen, die aufgeführten Beziehungen zwischen Ereignissen zu modellieren. Mit der Umsetzung des Ereignisstroms in das VTF3-Format wird die Nutzung des weit verbreiteten Analysewerkzeugs VAMPIR gewährleistet. Manche in diesem Kapitel vorgeschlagenen Modellierungshilfen, wie z.B. Abhängigkeiten, sind in VAMPIR bzw. im VTF3-Spurformat nicht verfügbar, sodass diese durch die vorhandenen Modellierungshilfen, wie z.B. Nachrichten, ausgedrückt werden müssen. Dies kann dazu führen, dass die in VAMPIR verfügbaren Statistiken über den Ereignisstrom verfälscht werden.

7.2 Geschlossene Visualisierung der RMA-Operationen

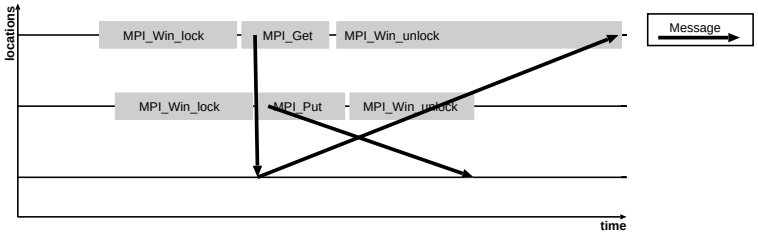
Wie in Kapitel 4.4.2 bereits erwähnt, ergibt die alleinige Modellierung einer Get-Operation durch die beiden Ereignistypen des Datentransfers, GET_1TS und GET_1TE, keine Verbindung zu deren Region. die den Datentransfer veranlasst hat. Erst die Erweiterung der Modellierung durch den zusätzlich eingeführten Ereignistyp GET_1TO stellt auch für das erweiterte Ereignismodell wieder eine Verbindung zu dieser Region her. Die Visualisierung des



(a) Get-Operation mit Synchronisation durch Fence



(b) Get-Operation mit allgemeiner Synchronisation mit aktivem Ziel



(c) Put- und Get-Operation mit Synchronisation mit passivem Ziel

Abbildung 7.1: Geschlossene Visualisierung der RMA-Operationen

Datentransfers in VAMPIR wird rein anhand der Ereignisse zum Transfer-Start und Transfer-Ende dargestellt. Das bedeutet, trotz der zusätzlichen Modellierung ist in der Visualisierung diese Information nicht dargestellt.

Durch das zusätzliche Ereignis entsteht eine erweiterte Ereigniskette für Get-Operationen der einseitigen Kommunikation:

1. Der Transfer wird am Ursprung der RMA-Operation veranlasst (transfer origin).
2. Der Transfer wird auf dem Zielprozess gestartet (transfer start).
3. Der Transfer wird durch den vollständigen Empfang der Daten auf dem Quellprozess beendet (transfer end).

Eine Visualisierung dieser Ereigniskette wäre wünschenswert. Da VAMPIR insbesondere für die Visualisierung von MPI-Programmen konzipiert wurde, sind die Modellierungshilfsmittel an die Konzepte des Nachrichtenaustauschs gebunden. Das bedeutet, die oben aufgeführte Ereigniskette ist durch die folgende Nachrichtensequenz modellierbar:

1. Der Ursprungsprozess sendet eine Nachricht der Größe 0 an den Zielprozess, um Daten aus dessen Speicherfenster anzufordern. Der Sendezeitpunkt liegt beim GET_1TO-Ereignis und der Empfangszeitpunkt wird durch das GET_1TS-Ereignis bestimmt.
2. Der Zielprozess der RMA-Operation sendet eine Nachricht mit den angeforderten Daten an den Ursprungsprozess der RMA-Operation.

Hierbei ist besonders zu betonen, dass es sich um eine reine Visualisierungshilfe für den Anwender handelt und hierdurch eine zusätzliche Nachricht in den zu visualisierenden Ereignisstrom eingefügt wird. Der Ereignisstrom wird in dem Sinne also verfälscht. Statistiken

über die im Programm versendeten Nachrichten werden damit ebenfalls beeinflusst. Durch das zugrunde liegende Ereignismodell ist gewährleistet, dass die Nachrichten niemals im Ereignisstrom zurücklaufen. Durch die Nachrichtengröße von 0 Bytes resultiert für diese Nachricht eine Bandbreite von 0 Byte/s. Davon würde eine globale Berechnung der Bandbreite beeinflusst. Die aus dieser Modellierung resultierenden Darstellungen der Datentransfers sind in Abbildung 7.1 dargestellt.

Vor- und Nachteile

Die implizite Verbindung innerhalb des Ereignisstroms ist nun zur Visualisierung genutzt. Durch die Einführung einer zusätzlichen Nachricht in den zu visualisierenden Ereignisstrom erhält der Anwender ein vollständig visuell verbundene Ereigniskette. Die verbesserte Visualisierung geht auf Kosten der globalen Leistungsstatistiken, welche durch die zusätzliche Nachricht beeinflusst werden. Der Anwender muss abwägen wie stark diese Beeinträchtigung im Vergleich zur gewonnenen Übersichtlichkeit wiegt. Durch eine rein lokale Betrachtung einer einzelnen Nachricht oder, bei einer Modellierung mit unterschiedlichen Nachrichten-Tags für die beiden Nachrichten einer Get-Operation, von Nachrichtengruppen, kann allerdings eine sinnvolle Bandbreite ermittelt werden. Das in Abbildung 7.1(c) dargestellte Szenario zeigt aber, dass der Datentransfer möglicherweise falsch dargestellt werden kann. Durch einen nicht blockierenden `MPI_Win_lock`-Aufruf ist es hier möglich, dass der Prozess mit der Get-Operation den Fensterinhalt *vor* der Put-Operation des anderen Prozesses zu erlangen scheint. Die Reihenfolge der Austrittsereignisse der `MPI_Win_unlock`-Aufrufe zeigt, dass die Put-Operation vor der Get-Operation abgeschlossen ist, und somit das Ergebnis der Get-Operation der vorher durch die Put-Operation gespeicherte Wert ist. Vorteil dieser Visualisierungshilfe ist, dass sie in Bezug auf KOJAK komplett in der Umsetzung der Ereignisspur auf das VTF3-Format getätigt werden kann, und keine Änderung des eigentlichen Ereignismodells verlangt.

7.3 Visualisierung des logischen Datentransfers

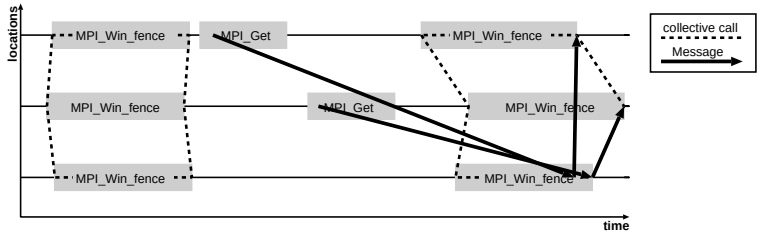
Wenn die Zielsetzung eine andere ist, kann die Visualisierung der Vorgänge auch sehr unterschiedlich zum wirklichen Datentransfer aussehen. Das folgende Visualisierungsmodell beruht auf der Annahme, dass in einem Cluster-basierten System Nachrichten erst zum spätestmöglichen Zeitpunkt gesendet werden, um Nachrichten gebündelt übertragen zu können. Dann kann nicht davon ausgegangen werden, dass der Datentransfer eine Get-Operation so schnell wie möglich gestartet wird, sondern vielmehr so lange wie möglich verzögert wird, um dann mit maximaler Effizienz übertragen zu werden. Das resultierende Visualisierungsmodell ist in Abbildung 7.2 angegeben. Die Zeitpunkte des Datentransfers einer Get-Operation orientieren sich dabei am folgenden Modell:

$$t_{Get\_TS} = min\left[\left(t_{SyncExit}^{Ursprung} - \varepsilon\right), t_{SyncExit}^{Ziel}\right]$$
$$t_{Get\_TE} = t_{SyncExit}^{Ursprung}$$

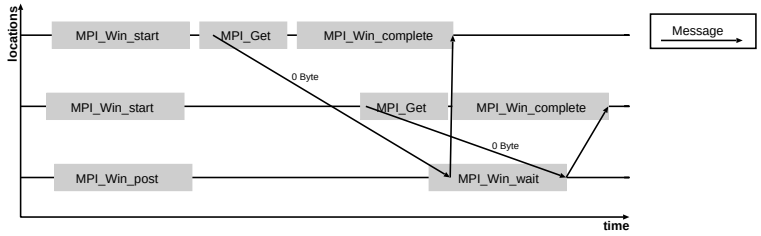
Bei der Synchronisation mit passivem Ziel ergeben sich direkt die folgenden Werte, da auf der Seite des Zielprozesses keine explizite Synchronisation aufgerufen wird.

$$t_{Get\_TS} = t_{SyncExit}^{Ursprung} - \varepsilon$$
$$t_{Get\_TE} = t_{SyncExit}^{Ursprung}$$

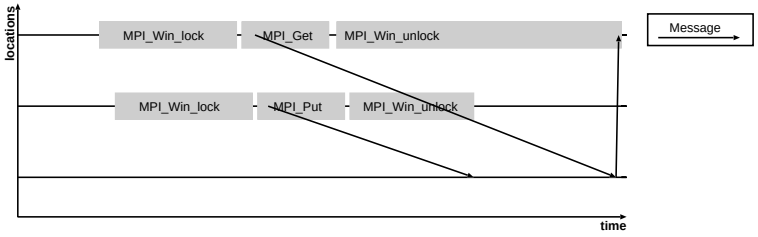
Um nicht einen zeitlosen Datentransfer modellieren zu müssen, wird bei der Berechnung des Transferbeginns der Zeitstempel, auf der Seite des Ursprungsprozesses, um ein sehr



(a) Get-Operation mit Synchronisation durch Fence



(b) Get-Operation mit allgemeiner Synchronisation mit aktivem Ziel



(c) Put- und Get-Operation mit Synchronisation mit passivem Ziel

Abbildung 7.2: Visualisierung des logischen Datentransfers

kleines ε verringert. ε darf nicht größer als die Differenz der Zeitstempel zwischen ENTER und EXIT-Ereignis sein. Dadurch ist gewährleistet, dass im zugrunde liegenden Ereignismodell bei einer chronologischen Ereignisreihenfolge keine zeitlich zurücklaufenden Nachrichten modelliert werden. D.h. Transferbeginn ist immer *vor* dem Transferende. Die unter Umständen sehr geringe zeitliche Differenz zwischen den beiden Zeitstempeln lässt die Bandbreite der modellierten Übertragung meist höher als physikalisch möglich erscheinen. Deshalb sind die auf diesem Modell beruhenden Daten statistisch nicht mehr auswertbar. Allerdings ist durch eine derartige Modellierung gewährleistet, dass die Datentransfers in der logisch korrekten Reihenfolge modelliert werden.

Vor- und Nachteile

Die korrekte Darstellung der logischen Abfolgen wird mit der Unbrauchbarkeit der Nachrichtenstatistiken erkauft. Falls der Anwender allerdings sein Programm auf der logischen Ebene nach Fehlern in der Abfolge der ausgetauschten Nachrichten überprüfen möchte, gibt dieses Modell ein besseres Bild, da der Fehler des anderen Visualisierungsmodells vermieden wird. Um diese Darstellung der Ereignisse zu erreichen, ist im Vergleich zum erweiterten Ereignismodell eine erneute Anpassung des zugrunde liegenden Ereignismodells nötig, da nun sowohl die Ereignisse der Typen GET_1TS als auch die Ereignisse der Typen GET_1TE nicht an der Stelle modelliert werden, an der sie aufgezeichnet werden können.

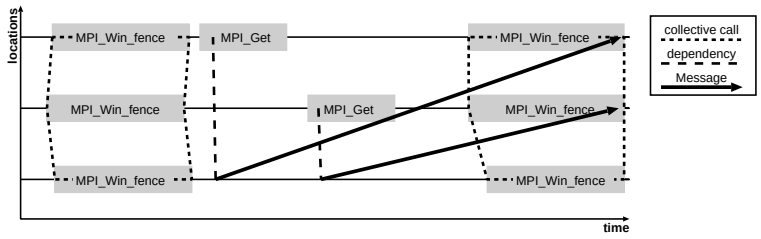
7.4 Visualisierung der Abhängigkeiten

Da die einseitige Kommunikation in MPI im Sinne der Abgeschlossenheit einer RMA-Operation stark von der umliegenden Synchronisation abhängt, kann es für die Anwender hilfreich sein, die bestehenden Abhängigkeiten zwischen den Synchronisationsfunktionen zu visualisieren. Die Offenheit des MPI-Standards erschwert in diesem Fall die Modellierung durch die zur Verfügung stehenden Mittel. Die Abhängigkeiten innerhalb der Synchronisation mit Fence sind durch die Möglichkeiten der Modellierung einer kollektiven Operation gegeben. Bei der allgemeinen Synchronisation mit aktivem Ziel jedoch, ist diese Art der Darstellung nicht praktikabel. Aus Leistungsgründen erlaubt der MPI-Standard eine komplette Zugriffsepoche zu puffern, bis die zugehörige Freigabeepoche begonnen hat. Das bedeutet, dass diese Epochen zeitlich so stark von einander verschieden seien dürfen, dass sie sich im Ereignisstrom nicht mehr überlappen. Da die Teilnahme an der Synchronisation für die in den Gruppen enthaltenen Prozesse bindend ist, liegt die Vermutung nahe, alle beteiligten Prozesse über die Modellierungshilfsmittel der kollektiven Operationen zu verbinden, da die dort genutzten Verbindungslinien im Ereignisstrom sowohl vor als auch zurücklaufen dürfen. Die Nutzung der Modellierungsmittel der kollektiven Operationen ist allerdings nicht praktikabel, da durch diese erlaubte Verzögerung eine Verzerrung der Verbindungslinien erzeugt wird. Verschiedene Zugriffs- und Freigabeepochen verhalten sich zusätzlich auch nicht wie kollektive Operationen, die eine wohldefinierte Reihenfolge auf allen beteiligten Prozessen haben müssen. Verschiedene Epochen können überlappen und in verschiedenen Reihenfolgen im Ereignisstrom auftreten. Das würde zu einem Bild führen, dass der üblichen Darstellung von kollektiven Operationen widerspricht.

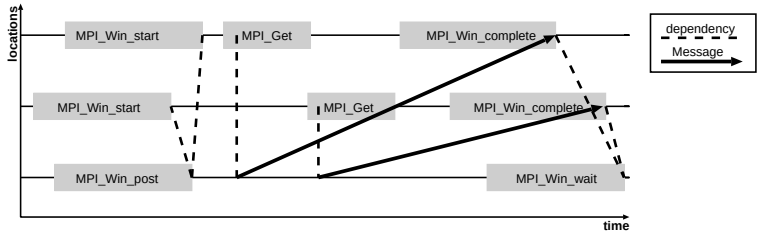
Eine Darstellung durch Nachrichten ist nur eingeschränkt möglich, da Nachrichten eine gerichtete Verbindung sind. Eine logische Abhängigkeit ist ebenfalls eine gerichtete Verbindung, wie z.B., dass eine Zugriffsepoche erst starten kann, wenn die Freigabeepoche gestartet ist. Aus Leistungsgründen ist eine Pufferung der gesendeten Nachrichten erlaubt, was ist in einer realen Implementierung dazu führen kann, dass das MPIWEXIT-Ereignis des `MPI_Win_start`-Aufrufs sowohl vor, als auch nach dem MPIWEXIT-Ereignis des `MPI_Win_post`-Aufrufs vorkommen kann. Es ist somit keine einheitliche Modellierung als gerichtete „ist-abhängig-von“-Beziehung möglich, da Nachrichten sowohl vorwärts als auch rückwärts laufen könnten, was dem reellen Verhalten einer Nachricht widerspricht. Es müssten neue Modellierungsmöglichkeiten im Spurformat bereitgestellt werden, um dies zu ermöglichen.

Aber nicht allein die Möglichkeiten der Visualisierung stoßen bei der Modellierung der Abhängigkeiten an ihre Grenzen. Das in dieser Arbeit benutzte Meta-Modell nutzt zur Herstellung von Abhängigkeiten vordergründig Zeigerattribute. Diese Zeigerattribute sind eine gerichtete Verbindung, mit der $n : 1$-Verbindungen modelliert werden können. D.h. mehrere Zeiger können auf das gleiche Ereignis zeigen. Ein Zeiger selbst kann aber immer nur auf maximal ein anderes Ereignis zeigen. Weitere Verbindungen und Beziehungen werden in der Darstellung des Modells nicht explizit aufgeführt, sondern in der Implementierung des Modells durch Hilfsdatenstrukturen ermöglicht. Ein Beispiel dafür sind die kollektiven Funktionen von MPI, die nur durch MPICEXIT- und MPIWCEXIT-Ereignisse modelliert werden, ohne untereinander mit Zeigerattributen verbunden zu sein. Dadurch dass die Zeigerattribute des Ereignismodells nur auf bereits geschehene Ereignisse verweisen dürfen, ist das Zeigermodell, sowie das Nachrichtenmodell, nicht praktikabel und eine Modellierung durch Hilfsstrukturen damit impliziert.

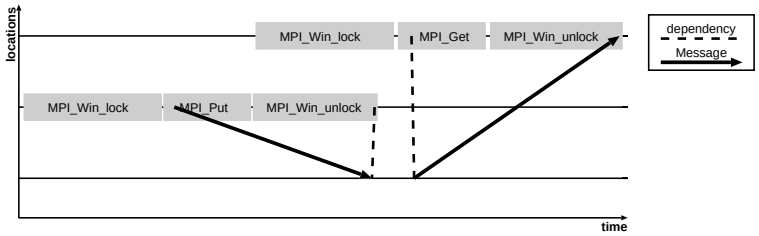
Durch ein eigenes Modellierungsinstrument für Abhängigkeiten werden allerdings trotz aller Hindernisse im Meta-Modell die Nachrichtenstatistiken und Visualisierungsmöglichkeiten erheblich entlastet. Abhängigkeiten müssen quasi-zeitlos modelliert werden und im Ereig-



(a) Synchronisation mit Fence



(b) allgemeine Synchronisation mit aktivem Ziel



(c) Synchronisation mit passivem Ziel

Abbildung 7.3: Visualisierung der Abhängigkeiten

nisstrom sowohl vor als auch zurück laufen dürfen. Dadurch ergeben sich zum Beispiel für die am Anfang erwähnte Modellierung geschlossene Darstellung der RMA-Operationen neue Möglichkeiten. Die Herstellung der visuellen Verbindung zur Get-Region kann nun durch eine Abhängigkeit ausgedrückt werden anstatt einer Nachricht. Im Modell würde dies bedeuten: „Der Start eines Datentransfers ist abhängig von der Erstellung einer virtuellen Anforderung am Transfer-Ursprung.“ In Abbildung 7.3 ist dies in einer möglichen Form der Visualisierung dargestellt.

Eine visuelle Verbindung der Fence-Aufrufe ist durch ihre Charakterisierung als kollektive Operationen bereits gegeben. Bis auf die Synchronisation mit passivem Ziel sind für die Put-Operation selbst keine weiteren Abhängigkeiten zu modellieren, da die End-Ereignisse der Datentransfers bereits bei den Synchronisationsfunktionen liegen, von denen sie abhängig sind. Im Falle der Synchronisation mit passivem Ziel, wie in Abbildung 7.3(c) dargestellt, wird von dem EXIT-Ereignis des `MPI_Win_unlock`-Aufrufs zum PUT_1TE-Ereignis eine Abhängigkeits-Verbindung modelliert, denn der Aufruf darf nicht an das Anwenderprogramm zurückkehren, bevor der Datentransfer nicht beendet ist. Bei der Modellierung der Get-Operationen kann die erste Nachricht, nun durch eine Abhängigkeit getauscht werden.

Vor- und Nachteile

Dieses Modell beschreibt durch die Abhängigkeiten genauer, welche Verbindungen zwischen den sonst voneinander losgelösten Regionen im Ereignisstrom existieren. Dies kann eine erhebliche Stütze für den Anwender bei der Zuordnung der Abläufe der einseitigen Kommunikation sein. Leider unterstützt eine aktuelle Version der Werkzeugumgebung VAMPIR keine

zusätzlichen Modellierungshilfsmittel, um diese Abhängigkeiten konfliktfrei darzustellen. Dies belässt den Anwender vor der Entscheidung entweder eine detailliertere Modellierung der Abhängigkeiten oder korrekte Aussagen über Nachrichtenstatistiken zu erhalten. Zusätzlich kann die Modellierung als Nachricht noch Fehlermeldungen verursachen, wenn, je nachdem wie die zu Grunde liegende Implementation gestaltet ist, Nachrichten im Ereignisstrom als rückwärts-laufend dargestellt werden. Eine Anpassung der Software in diesem Bereich der Modellierung ist somit wünschenswert, bevor weitere Unterstützung dieses Modells durch die tiefer liegende spurerzeugende Infrastruktur gegeben wird. Das zugrunde liegen Ereignismodell müsste zur Unterstützung dieser Abhängigkeiten ebenfalls erweitert werden.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Arbeit wurden Methoden zur ereignisbasierten Leistungsanalyse von Remote-Memory-Access Operationen erarbeitet und erläutert. Um die dafür nötigen Modellbildungen zu verstehen, wurde zunächst in Kapitel 2 eine Einführung in die Konzepte der ereignisbasierten Leistungsanalyse gegeben. Weiterhin wurde die Werkzeugumgebung KOJAK vorgestellt und in ihren, für die Analyse der einseitigen Kommunikation in MPI, relevanten Teilen erläutert. Die Leistungsanalyse von Kommunikations- und Ereignismustern, innerhalb eines parallelen Programms, setzt ein grundlegendes Verständnis der Vorgänge und Semantiken des Kommunikationskonzepts voraus. Deshalb wurde in Kapitel 3 eine Einführung in die einseitige Kommunikation mit MPI gegeben. Ein wichtiges Merkmal der einseitigen Kommunikation in MPI ist, dass die Synchronisation der Prozesse neben dem wechselseitigen Ausschluss beim Datenzugriff auch den Abschluss der RMA-Operationen bestimmt. Andere einseitige Kommunikationsbibliotheken, wie SHMEM, benötigen nur eine Synchronisation zum wechselseitigen Ausschluss. Diese feste Einbindung der Synchronisation, in die Kommunikationsschemata der einseitigen Kommunikation, bedingt eine starke Berücksichtigung dieser Funktionalität bei der Leistungsanalyse. Der MPI-Standard bietet drei mögliche Synchronisationsverfahren, die mit ihren Eigenschaften auf verschiedene Bedürfnisse der Interprozesskommunikation zugeschnitten sind. Diese Synchronisationsverfahren sind die Synchronisation mit Fence, die allgemeine Synchronisation mit aktivem Ziel sowie die Synchronisation mit passivem Ziel.

In Kapitel 4 wurden zwei Modelle entwickelt, die sich in der Leistungsanalyse der einseitigen Kommunikation bewähren müssen. Das vorgestellte Basismodell ermöglicht eine weitestgehende Aufzeichnungskompatibilität mit der Aufzeichnungsbibliothek VAMPIRTRACE. Dies eröffnet Anwendern die Möglichkeit, EPILOG als einzelnen Bestandteil aus KOJAK für die Aufzeichnung der Ereignisse des dynamischen Ablaufs ihrer Programme zu benutzen. Dadurch erhält der Anwender die zusätzliche Instrumentierung und Darstellung der Regionen der Synchronisationsfunktionen `MPI_Win_post`, `MPI_Win_wait`, `MPI_Win_test`, `MPI_Win_start` und `MPI_Win_complete`. Zusätzlich ergibt die Nutzung der Aufzeichnung den weiteren Vorteil, dass eine Konvertierung einer Ereignisspur in der Regel erheblich schneller ist, als ein Programmlauf für die Analyse durch KOJAK und ein weiterer für die Analyse durch VAMPIR.

Um die Schwächen des Basismodells zu minimieren und eine möglichst realitätsnahe Abbildung der Ereignisse zu gewährleisten, wurde das erweiterte Ereignismodell entwickelt. Dies benötigt eine komplexere Behandlung des Ereignisstroms bei der Zusammenführung der Einzelspuren zu einer globalen Spur, belohnt dies aber mit einer standardkonformen Abbildung der Ereignisse, unter Einbindung der im MPI-Standard beschriebenen Synchronisation. Mit Hilfe des erweiterten Modells können die dynamischen Abläufe des Anwenderprogramms genauer abgebildet werden und bieten für die Definition von Leistungsmerkmalen

und Leistungsproblemen eine geeignete Grundlage.

Weitergehend wurden zwei Modelle vorgestellt, die eine Visualisierung des Ereignisstroms für den Anwender in dem Maße unterstützen, dass die im MPI-Standard festgehaltenen logischen Verbindungen zwischen verschiedenen Ereignissen zusätzlich visualisiert werden können. Diese Modelle, die den Detaillierungsgrad des erweiterten Modells erhöhen, beseitigen im erweiterten Modell bestehende Schwächen in der Visualisierung. Im erweiterten Modell wird zum Beispiel bei einer Get-Operation aufgrund der Verschiebung der Datentransfereereignisse in die Synchronisation, der modellierte Datentransfer visuell von der Region gelöst. Der Anwender kann deshalb bei der Darstellung der Ereignisse in einem Zeitlinien-Diagramm keine intuitive Zuordnung der Regionen zu den aufgezeichneten Datentransfers vornehmen, sobald mehrere gleichzeitige Get-Operationen für den Prozess aufgezeichnet wurden. Eine logische Verbindung zwischen dem Enter-Ereignis der Get-Region und dem Start-Ereignis des Datentransfers stellt diese intuitive Verbindung für den Anwender wieder her. Diese kann durch eine zusätzliche Nachricht modelliert werden, die während der Spurkonvertierung erzeugt wird und somit nur in der visualisierten Spur explizit existiert. Weitere Ansätze zur Visualisierung und damit zur Unterstützung der manuellen Analyse wurden in Kapitel 7 gegeben. Diese beinhalten allerdings Änderungen des zugrunde liegenden Ereignismodells oder Änderungen in der Visualisierungssoftware.

In Kapitel 5 wurden die nötigen Änderungen in den Programmteilen von KOJAK erläutert, um das Basismodell und das erweiterte Modell voll zu unterstützen. An dieser Stelle sind noch Möglichkeiten zur Optimierung des Verarbeitungsprozesses bei der Erstellung der Gesamtspur gegeben. Es wurden in Kapitel 6 Leistungsmerkmale und Leistungsprobleme beschrieben, welche viele der, in der einseitigen Kommunikation möglichen, Muster zur ineffizienten Kommunikation aufzeigen können. Die bevorstehende stärkere Nutzung dieses Kommunikationskonzepts in produktionsfähigen Anwendungen wird zeigen, ob zu diesen Leistungsmerkmalen noch weitere hinzukommen. Aufgrund der limitierten Zeit wurden nicht alle in dieser Arbeit beschriebenen Leistungsmerkmale implementiert. Der Fokus lag auf der theoretischen Entwicklung der Modelle und die Implementation der dafür notwendigen Infrastruktur.

Die Arbeiten an der Implementierung der Ereignismodell wurden in einem separaten Zweig der KOJAK-Quellen realisiert. Während der Entwicklung sind in den Hauptstrang weitere Ergänzungen und Korrekturen eingeflossen. Eine Zusammenführung des Zweiges mit dem Hauptstrang der KOJAK-Quellen muss noch vollzogen werden, damit die Funktionalität der Analyse in die nächste Release von KOJAK aufgenommen werden kann.

Weitere Arbeiten an der Analyse der einseitigen Kommunikation beinhalten die Implementierung der entsprechend vorgeschlagenen Ereignistypen, Zustände und Zeigerattribute in EARL und die darauf aufbauende Definition der entwickelten Leistungsmerkmale in EXPERT. Zusätzlich ist die Arbeit an einer Lösung der korrekten Modellierung einer Sperrzuweisung nötig, um die Analyse in diesem Bereich zu erleichtern und weitere Leistungsmerkmale definieren zu können.

Anhang A

Glossar

Begriff	Übersetzung	Beschreibung
Anforderung	request	Datenstruktur, die bei der nicht-blockierenden Punkt-zu-Punkt-Kommunikation in MPI benötigt wird. Über die Anforderung kann das Anwenderprogramm den aktuellen Status der angeforderten Datenübertragung ermitteln.
Aufrufpfad	callpath	Liste von ↑Regionen, die die relative Schachtelung zum Hauptprogramm angibt.
Aufrufstapel	callstack	Kellerspeicher, der die Informationen über die betretenen ↑Regionen aufnimmt. Die enthaltenen Regionen bilden den ↑Aufrufpfad.
Definitionseintrag	definition record	Eintrag am Anfang einer EPILOG-↑Spur, der einem Bezeichner verschiedene Attributwerte zuordnet. Die Attribute unterscheiden sich zwischen verschiedene Defininitionseintragstypen. Definitionseinträge minimieren redundant gespeicherte Daten innerhalb der Ereigniseinträge der EPILOG-Spur.
einfaches Leistungsmerkmal	profiling patterns	Leistungsmerkmale, die eine auf Profiling-Daten basierende Bewertung verschiedener Regionsgruppen enthalten.
einseitige Kommunikation	one-sided / single-sided communication	Kommunikationsparadigma, bei dem alle Parameter des Datenaustauschs von einem der beiden Kommunikationspartner bestimmt werden, dem ↑Ursprungsprozess.

Begriff	Übersetzung	Beschreibung
entferntes Ereignis	remote event	↑Ereignis während der Messphase, welches ein Ereignis auf einem anderen Prozess kennzeichnet, da dieser das Ereignis zur Laufzeit nicht selbst generieren kann.
entfernter Speicherzugriff	remote memory access	lesender oder schreibender Zugriff eines Prozesses auf einen Speicherbereich eines weiteren Prozesses
Ereignis	event	Ein Ereignis ist eine Instanz eines ↑Ereignistyps, welche durch die Werte der beinhalteten Attribute eindeutig identifizierbar wird. Ein Ereignis hat keine zeitliche Ausdehnung.
Ereignisgeschichte	event history	Die chronologische Abfolge von ↑Ereignissen. Sie definiert einen modellierten Prozesszustand.
Ereignismodell	event modell	Beschreibung der Hilfsmittel und deren Einsatz bei der Modellierung realer Vorgänge.
Ereignisspur	event trace	Datei, welche einen ↑Ereignisstrom enthält.
Ereignisstrom	event stream	Chronologisch angeordnete Reihenfolge von ↑Ereignissen eines Programmablaufs.
Ereignistyp	event type	Menge von Attributen, die Informationen über einen, für die Leistungsanalyse wichtigen, Zeitpunkt innerhalb des Anwendungsprogramms beinhaltet.
Exklusive Sperre	exclusive lock	Sperre, die keinen weiteren Lese- oder Schreibzugriff auf die gesperrten Daten zulässt.
Freigabeepoche	exposure epoch	Zeitraum, in der ein ↑Zielprozess ↑RMA-Operationen auf eines seiner ↑RMA-Fenster zulässt.
Gemeinsame Sperre	shared lock	Sperre, die weitere Lesezugriffe auf die gesperrten Daten zulässt. Sie kann nur gesetzt werden, wenn der Prozess selbst reine Lesezugriffe während der Sperrzeit ausführt.

Begriff	Übersetzung	Beschreibung
Get-Operation	get operation	RMA-Operation, die Daten aus einem ↑RMA-Speicherfenster bezieht. In MPI ist dies nur die Funktion `MPI_Get`.
Instrumentierung	instrumentation	Einfügen von zusätzlichen Instruktionen zur Messung eines Programmabschnitts.
Leistungsanalyse-zyklus	performance analysis cycle	Vorgehensweise bei der Leistungsanalyse eines Programms. Er besteht aus Instrumentierungsphase, Messphase, Analysephase, Präsentationsphase und Optimierungsphase.
Leistungsengpass	performance bottleneck	↑Leistungsproblem eines spezifischen Programms mit den stärksten Auswirkungen.
Leistungsmerkmal	performance property	Eigenschaft eines Programms, das Einfluss auf die Leistung nimmt.
Leistungsproblem	performance problem	↑Leistungsmerkmal mit negativem Einfluss auf die Leistung des Programms.
musterbasiertes Leistungsmerkmal	complex pattern	↑Leistungsmerkmal, welches durch die Beziehung zwischen verschiedenen Ereignissen gebildet wird.
Nachrichtenaustausch	message passing	Kommunikation und Datenaustausch über Nachrichten.
Profiling	profiling	Sammeln von Laufzeitinformationen in summarischer Form.
Put-Operation	put operation	↑RMA-Operation, die Daten vom ↑Ursprung zum ↑Ziel in das angegebene ↑Speicherfenster überträgt. Dazu gehören die Aufrufe von `MPI_Put` und `MPI_Accumulate`.
Quelle, Quellprozess	source	Die Quelle der Daten einer Datenübertragung.
Region	region	Sequenz von Instruktionen, die als zusammenhängend modelliert werden sollen. Regionen werden durch ein Eintritts- und ein Austrittsereignis beschrieben.

Begriff	Übersetzung	Beschreibung
RMA-Fenster	rma window	Speicherbereich, der für die ↑einseitige Kommunikation für den Datenaustausch durch die Funktion `MPI_Win_create` vorbereitet wurde.
RMA-Operation	rma operation	Ein Aufruf von `MPI_Put`, `MPI_Get` oder `MPI_Accumulate`.
RMA-Speicherfenster	rma window	↑RMA-Fenster.
Sampling	sampling	Methode zur Erfassung von Leistungsdaten an anwendungsextern gesteuerten Unterbrechungspunkten eines Programmlaufs.
schwache Synchronisation	weak synchronisation	Synchronisationsart, bei der durch Pufferung der ausgetauschten Nachrichten eine Verschiebung der ↑Zugriffs- und ↑Freigabeepoche ermöglicht wird.
Spur	trace	↑Ereignisspur.
starke Synchronisation	strong synchronization	Synchronisationsart, bei der die Reihenfolge der ↑Zugriffs- und ↑Freigabeepochen die zeitlich logisch korrekte Abfolge einhält.
Ursprung, Ursprungsprozess	origin	Prozess, welcher die ↑RMA-Operation ausführt und somit der Prozess, der in der RMA-Operation auf ein ↑RMA-Fenster zugreift.
Synchronisation mit aktivem Ziel	active target synchronisation	Synchronisation der einseitigen Kommunikation durch explizite Aufrufe an ↑Ursprung *und* ↑Ziel. In MPI sind dies die kollektive Funktion `MPI_Win_fence` sowie die Funktionen `MPI_Win_post`, `MPI_Win_wait`, `MPI_Win_start` und `MPI_Win_complete`.
Synchronisation mit passivem Ziel	passive target synchronisation	Synchronisation der einseitigen Kommunikation ohne expliziten Aufruf am ↑Ziel. Bei MPI sind dies die Funktionen `MPI_Win_lock` und `MPI_Win_unlock`

Begriff	Übersetzung	Beschreibung
Tracing	tracing	Methode der Leistungsdatenerfassung durch Aufzeichnung von ↑Ereignisströmen.
Versatz	displacement	Gibt bei Kommunikationsfunktionen eine Verschiebung zur Basisadresse des Kommunikationspuffers oder ↑Speicherfensters an.
Wrapper	wrapper	Kapselung einer Funktion, die vor oder nach dem Aufruf der Funktion zusätzliche Instruktionen enthält.
Zeigerattribut	pointer attribute	Attribut eines ↑Ereignistyps des ↑Ereignismodells, mit dem Beziehungen zu anderen ↑Ereignissen im ↑Ereignisstrom hergestellt werden. Zeigerattribute können nur auf bereits vergangene Ereignisse zeigen. Sie weisen somit immer auf Ereignisse der ↑Ereignisgeschichte.
Ziel, Zielprozess	target	Prozess, welcher das ↑RMA-Fenster bereitstellt. Nicht zu verwechseln mit dem ↑Ziel einer Datenübertragung.
Ziel einer Datenübertragung	destination	Prozess der Daten eines Datentransfers empfängt.
Zugriffsepoche	access epoch	Zeitraum, in der ein ↑Ursprungsprozess auf ein ↑RMA-Fenster zugreift.

Literaturverzeichnis

[1] OpenMP Architecture Review Board. The OpenMP Application Programming Interface, 2002.
http://www.openmp.org/.

[2] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. In *2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing, SHPSEC-PACT03*, 2003.
http://www.cs.berkeley.edu/~bonachea/upc/bonachea-duell-mpi.pdf.

[3] Pallas GmbH. Vampir/vampirtrace performance analysis of parallel programs, 2004.
http://www.pallas.com/e/products/vampir/index.htm.

[4] F. Hoßfeld, K. Solchenbach, C. Bischof, and W. Nagel. Gekoppelte SMP-Systeme im wissenschaftlich-technischen Hochleistungsrechnen (GoSMP). Technical report, Forschungszentrum Jülich, Pallas GmbH, RWTH Aachen, TU Dresden, 2000.

[5] Glenn Luecke and Wei Hu. Evaluating the Performance of MPI-2 One-Sided Routines on a Cray SV1. Technical report, Iowa State University, 2002.

[6] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, B. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer: Special issue on performance evaluation tools for parallel and distributed computer systems*, November 1995.

[7] Bernd Mohr and Felix Wolf. KOJAK - Kit for Objective Judgement And Knowledge-based detection of performance bottlenecks, 2000.
http://www.fz-juelich.de/kojak/.

[8] Kathryn Mohror and Karen L. Karavanic. Performance Tool Support for MPI-2 on Linux. In *SC2004: Bridging Communities. Pittsburgh, PA*, November 2004.

[9] Message Passing Interface Forum MPIF. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennesse, Knoxville, 1994.
http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[10] Message Passing Interface Forum MPIF. *MPI-2: Extensions to the Message-Passing Interface*. University of Tennessee, Knoxville, 1996.
http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[11] Message Passing Interface Forum MPIF. *One-Sided Communication – Introduction*, chapter 6.1, page 109. In [10], 1996.
http://www.mpi-forum.org/docs/mpi-20-html/mpi-report.html.

[12] Message Passing Interface Forum MPIF. *Semantics and Correctnes*, chapter 6.7, page 138. In [10], 1996.
http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[13] W. Putman, J. Chern, S. Lin, W. Sawyer, and B. Shen. Modeling the Earth's Atmosphere. In *SC2002: From Terrabytes to Insights. Baltimore, MD*, November 2002.
http://www.nas.nasa.gov/About/Media/SC02/Goddard/goddard.html.

[14] Rolf Rabenseifner. HLRS - Höchstleistungsrechenzentrum Stuttgart, Persönliche Gespräche.

[15] Rolf Rabenseifner. *Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen.* PhD thesis, Universität Stuttgart, März 2000.

[16] PAPI Team. The Performance Aapplication Pprogramming Iinterface. Technical report, Innovative Computing Laboratory at University Tennessee, Knoxville, 2004. `http://icl.cs.utk.edu/papi/`.

[17] Felix Wolf. EARL - Eine programmierbare Umgebung zur Bewertung paralleler Prozesse auf Message-Passing-Systemen. Diplomarbeit, RWTH Aachen, Forschungszentrum Jülich, Juni 1998.

[18] Felix Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes.* PhD thesis, RWTH Aachen, December 2002.

[19] Felix Wolf. EARL — API Documentation. Technical Report ICL-UT-04-03, University of Tennessee, Forschungszentrum Jülich, Oktober 2004.

[20] Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *Journal of Systems Architecture: Evolutions in parallel distributed and network-based processing*, 49(10-11):421–439, November 2003.

[21] Felix Wolf and Bernd Mohr. EPILOG Binary Trace-Data Format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, Mai 2004.